US009436435B2

(12) **United States Patent**
Wolrich et al.

(10) **Patent No.: US 9,436,435 B2**
(45) **Date of Patent: Sep. 6, 2016**

(54) **APPARATUS AND METHOD FOR VECTOR INSTRUCTIONS FOR LARGE INTEGER ARITHMETIC**

(75) Inventors: **Gilbert M. Wolrich**, Framingham, MA (US); **Kirk S. Yap**, Framingham, MA (US); **James D. Guilford**, Northborough, MA (US); **Erdinc Ozturk**, Istonbul (TR); **Vinodh Gopal**, Westborough, MA (US); **Wajdi K. Feghali**, Boston, MA (US); **Sean M. Gulley**, Boston, MA (US); **Martin G. Dixon**, Portland, OR (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 133 days.

(21) Appl. No.: **13/996,529**

(22) PCT Filed: **Dec. 23, 2011**

(86) PCT No.: **PCT/US2011/067165**

§ 371 (c)(1),
(2), (4) Date: **Feb. 26, 2014**

(87) PCT Pub. No.: **WO2013/095629**

PCT Pub. Date: **Jun. 27, 2013**

(65) **Prior Publication Data**

US 2014/0164467 A1 Jun. 12, 2014

(51) **Int. Cl.**

G06F 7/525	(2006.01)
G06F 7/57	(2006.01)
G06F 9/30	(2006.01)
G06F 9/38	(2006.01)

(52) **U.S. Cl.**

CPC ***G06F 7/57*** (2013.01); ***G06F 9/3001*** (2013.01); ***G06F 9/30036*** (2013.01); ***G06F 9/3893*** (2013.01)

(58) **Field of Classification Search**

None

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,121,431 A 6/1992 Wiener
5,457,804 A * 10/1995 Ohtomo G06F 7/5324 708/523

(Continued)

FOREIGN PATENT DOCUMENTS

CN 101271570 A 9/2008

OTHER PUBLICATIONS

PCT Notification concerning Transmittal of International Preliminary Report on Patentability (Chapter I of the Patent Coopertation Treaty) for PCT Counterpart Application No. PCT/US2011/067165, 6 pgs., (Jul. 3, 2014).

(Continued)

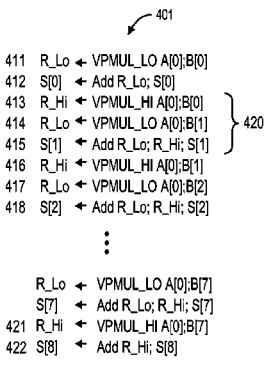
Primary Examiner — David H Malzahn

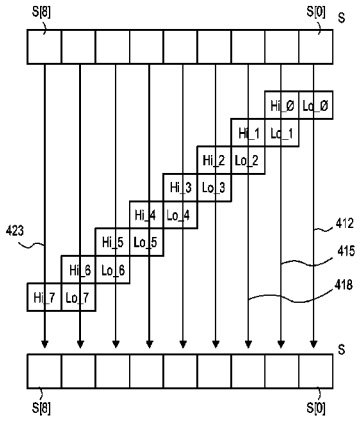
(74) *Attorney, Agent, or Firm* — Nicholson De Vos Webster & Elliott, LLP

(57) **ABSTRACT**

An apparatus is described that includes a semiconductor chip having an instruction execution pipeline having one or more execution units with respective logic circuitry to: a) execute a first instruction that multiplies a first input operand and a second input operand and presents a lower portion of the result, where, the first and second input operands are respective elements of first and second input vectors; b) execute a second instruction that multiplies a first input operand and a second input operand and presents an upper portion of the result, where, the first and second input operands are respective elements of first and second input vectors; and, c) execute an add instruction where a carry term of the add instruction's adding is recorded in a mask register.

22 Claims, 26 Drawing Sheets





(56) **References Cited**

U.S. PATENT DOCUMENTS

6,370,559	B1	4/2002	Hoffman et al.	
6,530,011	B1 *	3/2003	Choquette	G06F 7/483 708/236
6,598,064	B1	7/2003	Green	
8,019,805	B1 *	9/2011	Sarma	G06F 7/4876 708/503
8,549,264	B2	10/2013	Gopal et al.	
2006/0253520	A1 *	11/2006	Tran	G06F 7/5318 708/607
2006/0253522	A1	11/2006	Dror	
2014/0164467	A1 *	6/2014	Wolrich	G06F 9/30036 708/523

OTHER PUBLICATIONS

"Notification of Transmittal of the International Search Report and the Written Opinion of the International Searching Authority", Application No. PCT/US2011/067165, mailed Aug. 28, 2012, 9 pages.

Office action with English translation from Taiwan Patent Application No. 101148095, mailed Feb. 10, 2015, 4 pages.

Office action with English translation from Chinese Patent Application No. 201180075776.4, mailed Nov. 18, 2015, 19 pages.

* cited by examiner

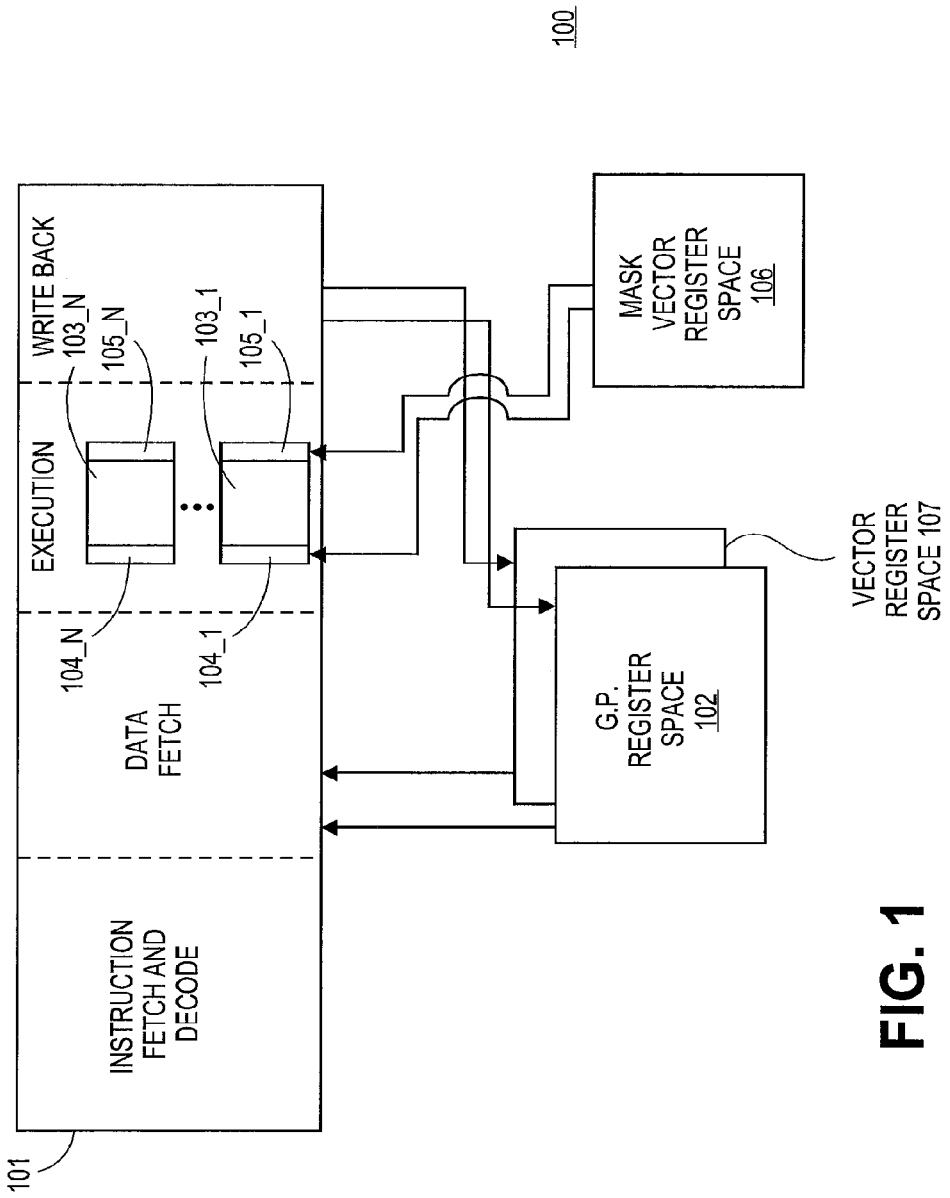

100
101
INSTRUCTION FETCH AND DECODE
DATA FETCH
EXECUTION
WRITE BACK
104_N
104_1
103_N
105_N
103_1
105_1
G.P. REGISTER SPACE 102
MASK VECTOR REGISTER SPACE 106
VECTOR REGISTER SPACE 107

FIG. 1

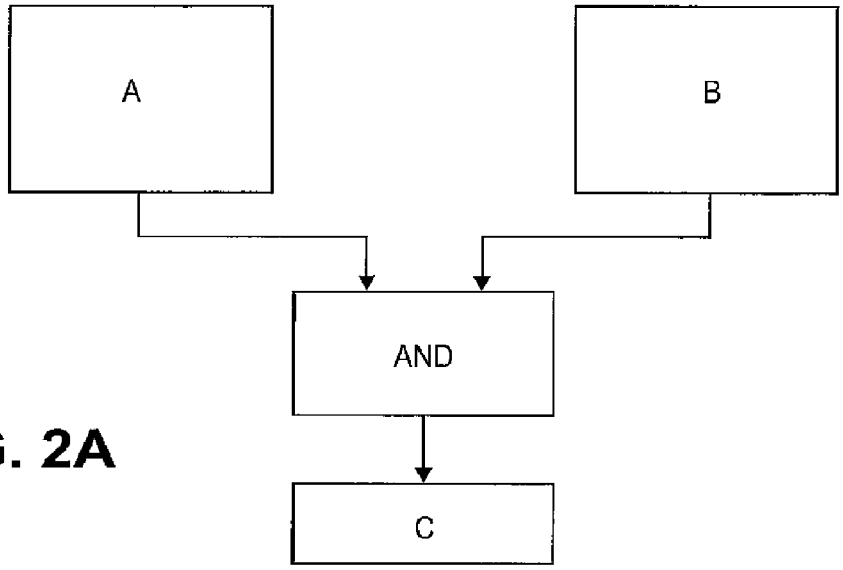


FIG. 2A

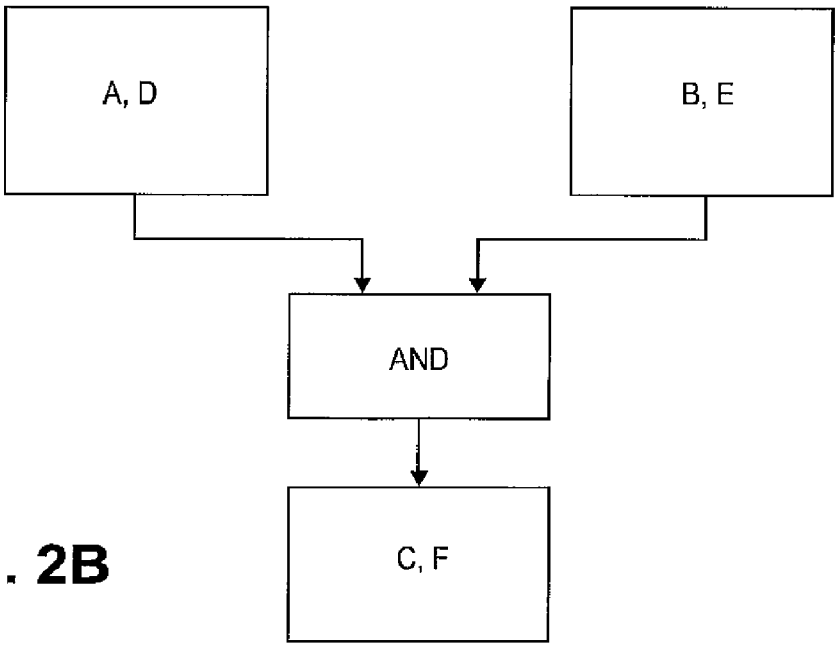


FIG. 2B

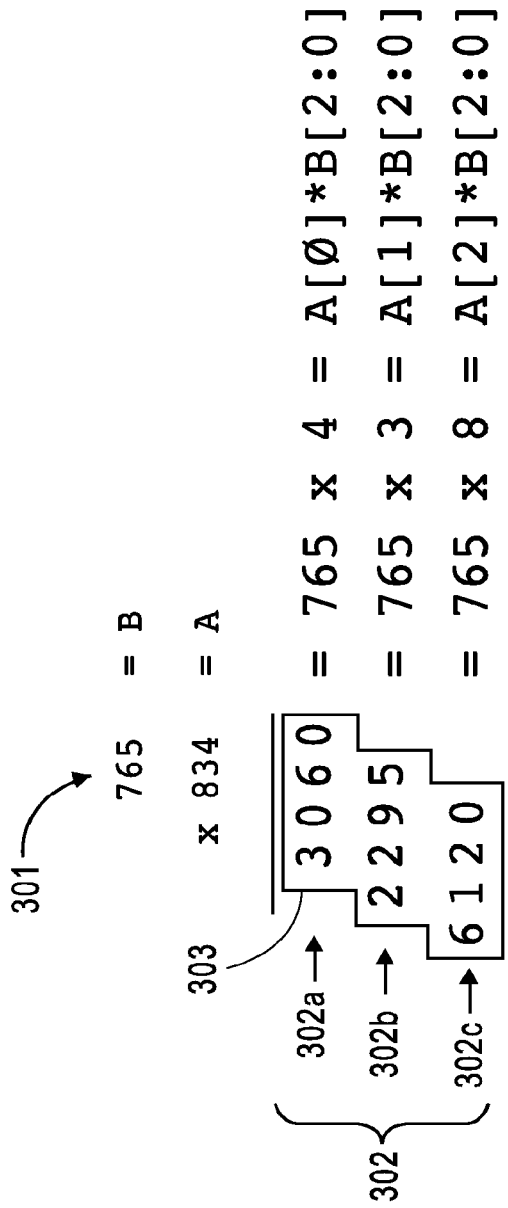

765 = B
x 834 = A
3060 = 765 x 4 = A[Ø]*B[2:0]
2295 = 765 x 3 = A[1]*B[2:0]
6120 = 765 x 8 = A[2]*B[2:0]
638010
301
303
302a
302b
302c
302

FIG. 3A

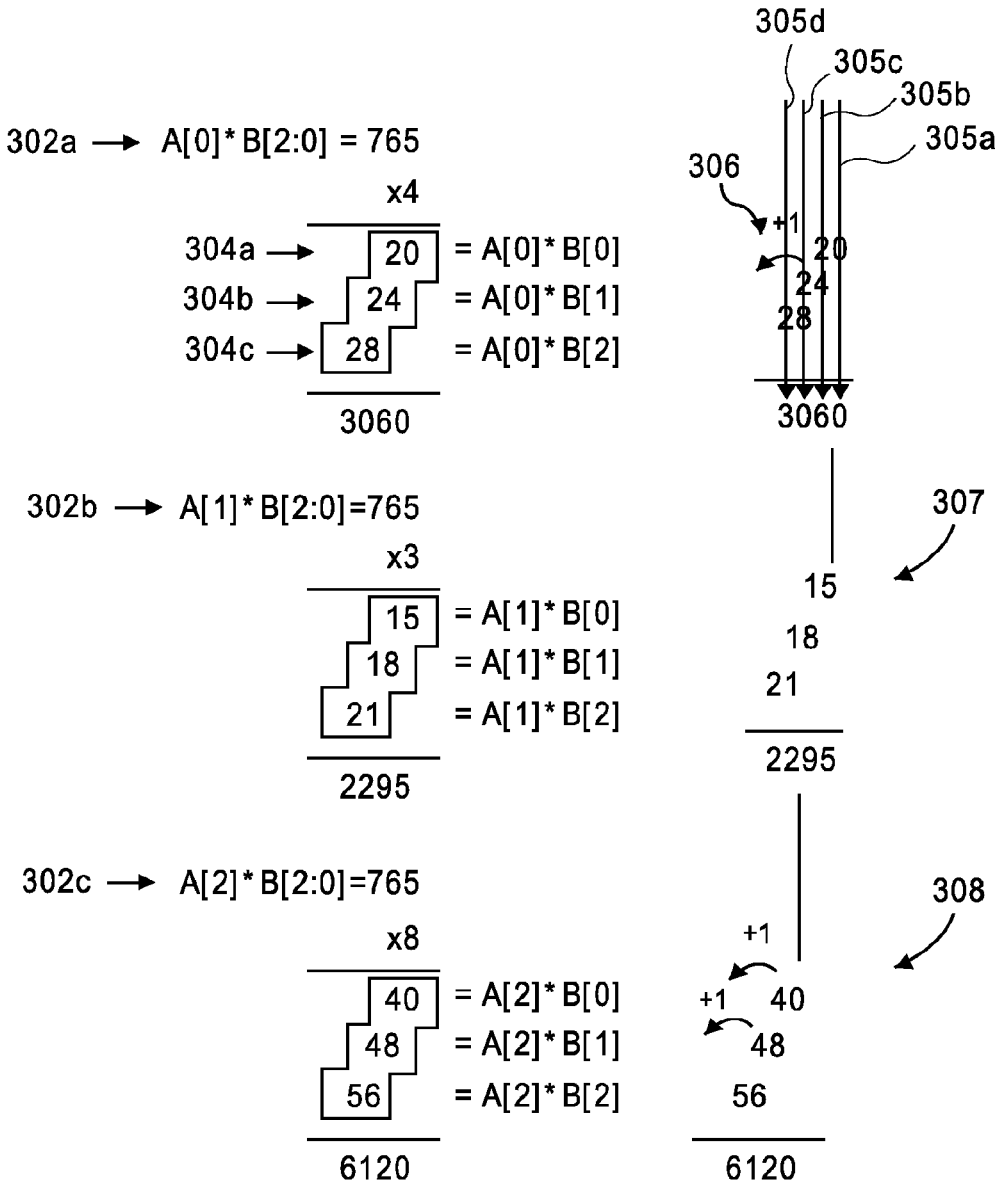

302a → A[0]*B[2:0] = 765
x4
304a → 20 = A[0]*B[0]
304b → 24 = A[0]*B[1]
304c → 28 = A[0]*B[2]
3060
302b → A[1]*B[2:0]=765
x3
15 = A[1]*B[0]
18 = A[1]*B[1]
21 = A[1]*B[2]
2295
302c → A[2]*B[2:0]=765
x8
40 = A[2]*B[0]
48 = A[2]*B[1]
56 = A[2]*B[2]
6120
305d
305c
305b
305a
306
+1
20
24
28
3060
307
15
18
21
2295
308
+1
+1
40
48
56
6120

FIG. 3B

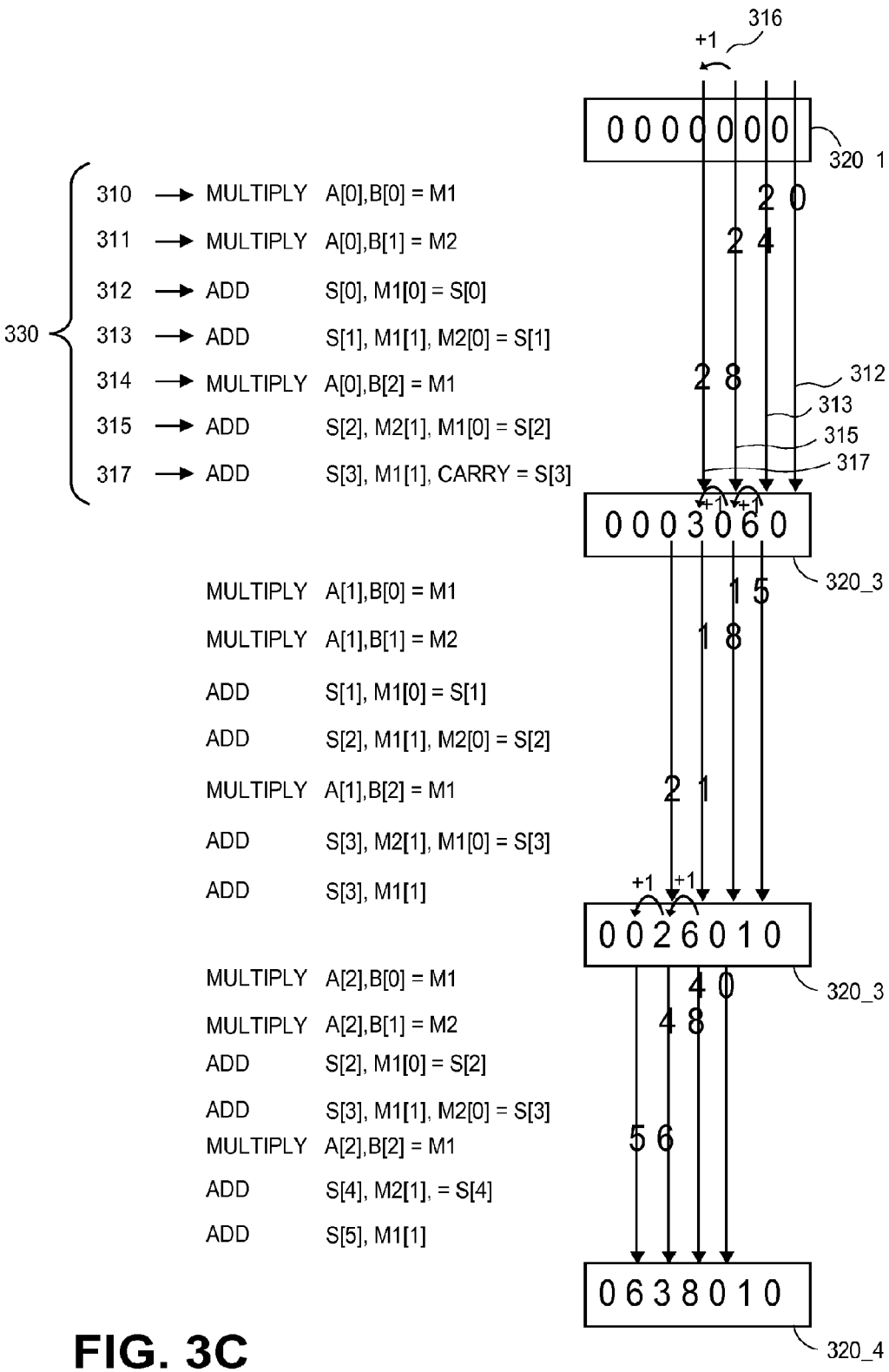
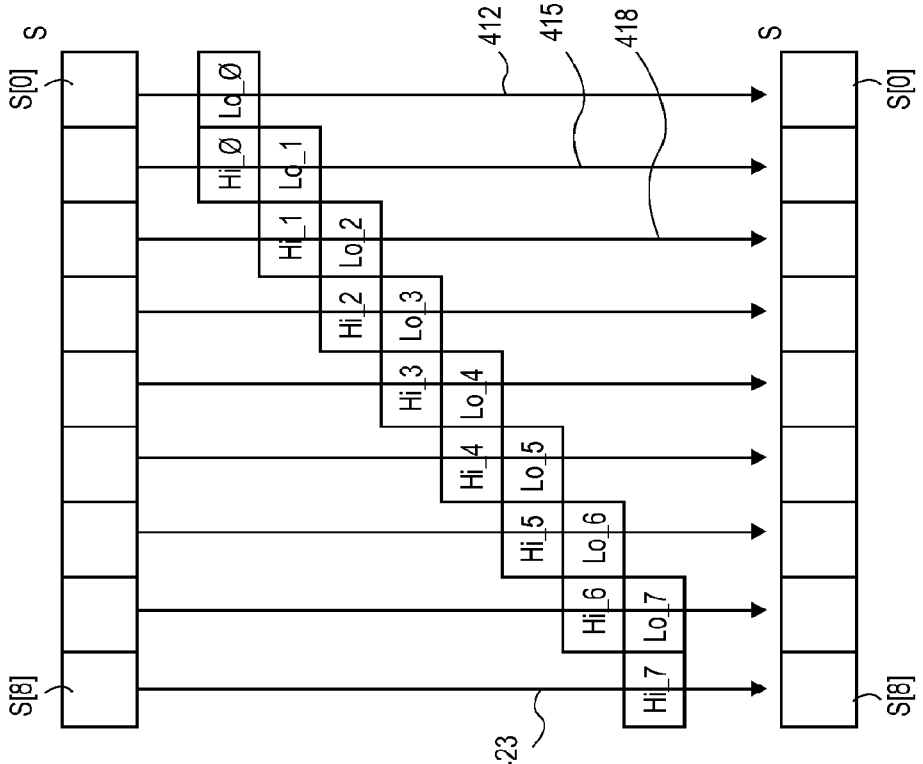

316
+1
0000000
320_1
20
24
28
312
313
315
317
+1 +1
0003060
320_3
15
18
21
+1 +1
0026010
320_3
40
48
56
0638010
320_4
330
310 MULTIPLY A[0],B[0] = M1
311 MULTIPLY A[0],B[1] = M2
312 ADD S[0], M1[0] = S[0]
313 ADD S[1], M1[1], M2[0] = S[1]
314 MULTIPLY A[0],B[2] = M1
315 ADD S[2], M2[1], M1[0] = S[2]
317 ADD S[3], M1[1], CARRY = S[3]
MULTIPLY A[1],B[0] = M1
MULTIPLY A[1],B[1] = M2
ADD S[1], M1[0] = S[1]
ADD S[2], M1[1], M2[0] = S[2]
MULTIPLY A[1],B[2] = M1
ADD S[3], M2[1], M1[0] = S[3]
ADD S[3], M1[1]
MULTIPLY A[2],B[0] = M1
MULTIPLY A[2],B[1] = M2
ADD S[2], M1[0] = S[2]
ADD S[3], M1[1], M2[0] = S[3]
MULTIPLY A[2],B[2] = M1
ADD S[4], M2[1], = S[4]
ADD S[5], M1[1]

FIG. 3C


S
S[0]
S[8]
Hi_Ø
Lo_Ø
Lo_1
Hi_1
Lo_2
Hi_2
Lo_3
Hi_3
Lo_4
Hi_4
Lo_5
Hi_5
Lo_6
Hi_6
Lo_7
Hi_7
412
415
418
423
S
S[0]
S[8]


401
420
411 R_Lo ← VPMUL_LO A[0];B[0]
412 S[0] ← Add R_Lo; S[0]
413 R_Hi ← VPMUL_HI A[0];B[0]
414 R_Lo ← VPMUL_LO A[0];B[1]
415 S[1] ← Add R_Lo; R_Hi; S[1]
416 R_Hi ← VPMUL_HI A[0];B[1]
417 R_Lo ← VPMUL_LO A[0];B[2]
418 S[2] ← Add R_Lo; R_Hi; S[2]
R_Lo ← VPMUL_LO A[0];B[7]
S[7] ← Add R_Lo; R_Hi; S[7]
421 R_Hi ← VPMUL_HI A[0];B[7]
422 S[8] ← Add R_Hi; S[8]

FIG. 4A

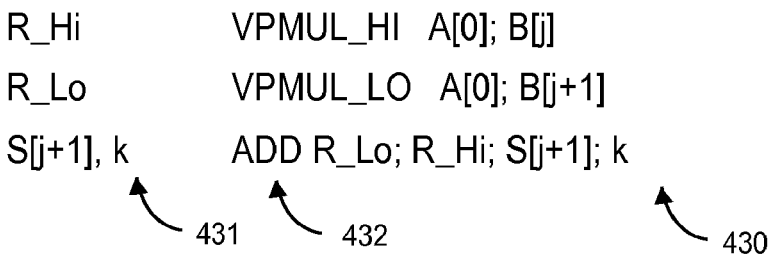

R_Hi VPMUL_HI A[0]; B[j]
R_Lo VPMUL_LO A[0]; B[j+1]
S[j+1], k ADD R_Lo; R_Hi; S[j+1]; k
431
432
430

FIG. 4B

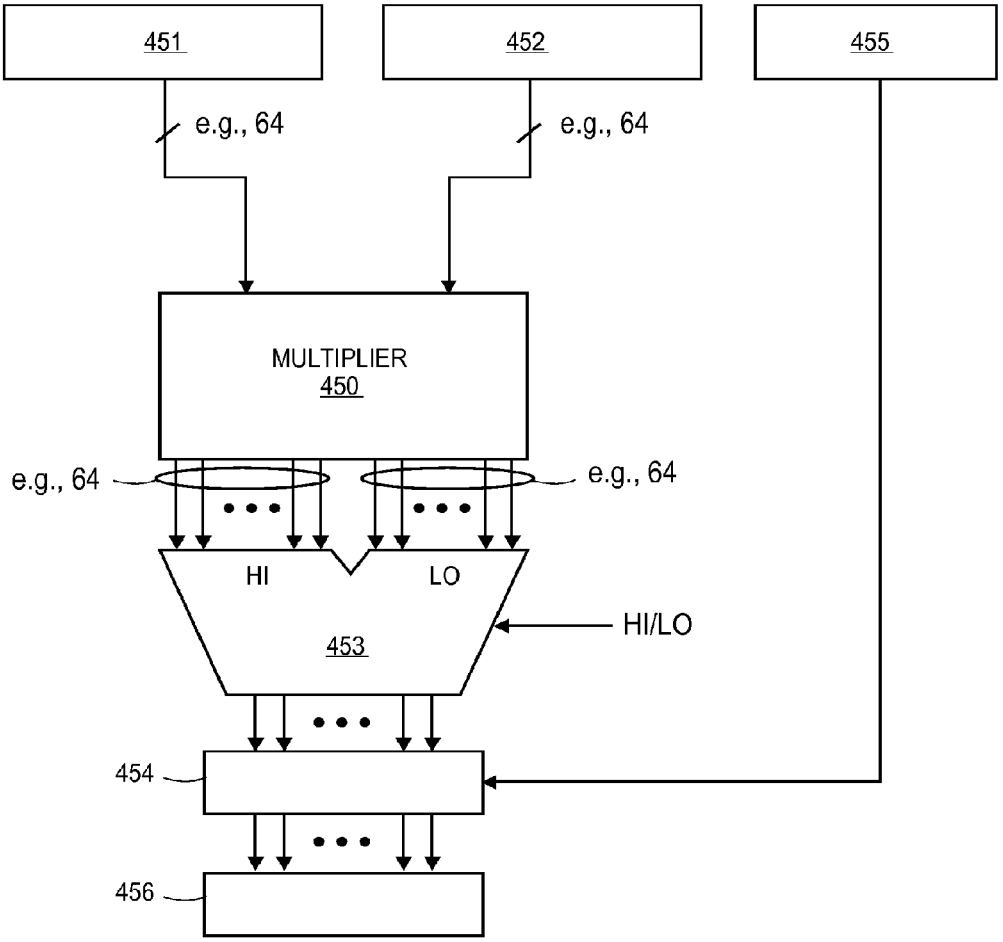

451
452
455
e.g., 64
e.g., 64
MULTIPLIER
450
e.g., 64
e.g., 64
HI
LO
HI/LO
453
454
456

FIG. 4C

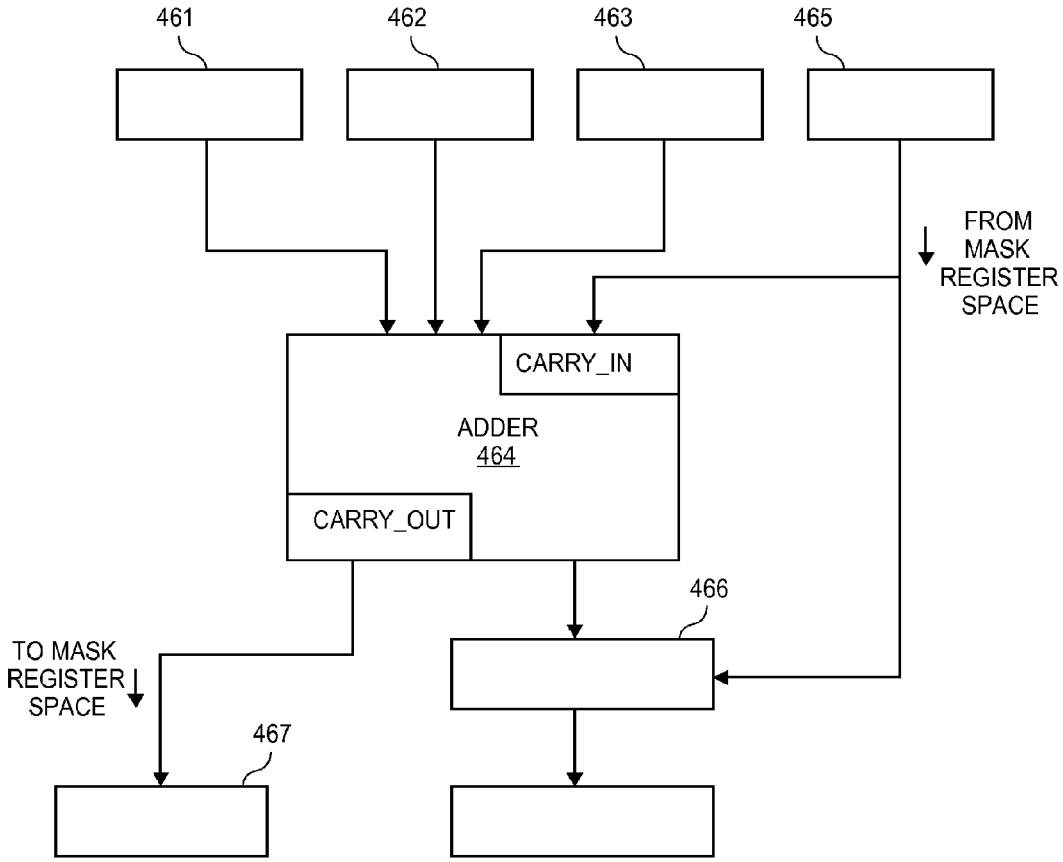

461
462
463
465
FROM MASK REGISTER SPACE
CARRY_IN
ADDER
464
CARRY_OUT
466
TO MASK REGISTER SPACE
467

FIG. 4D

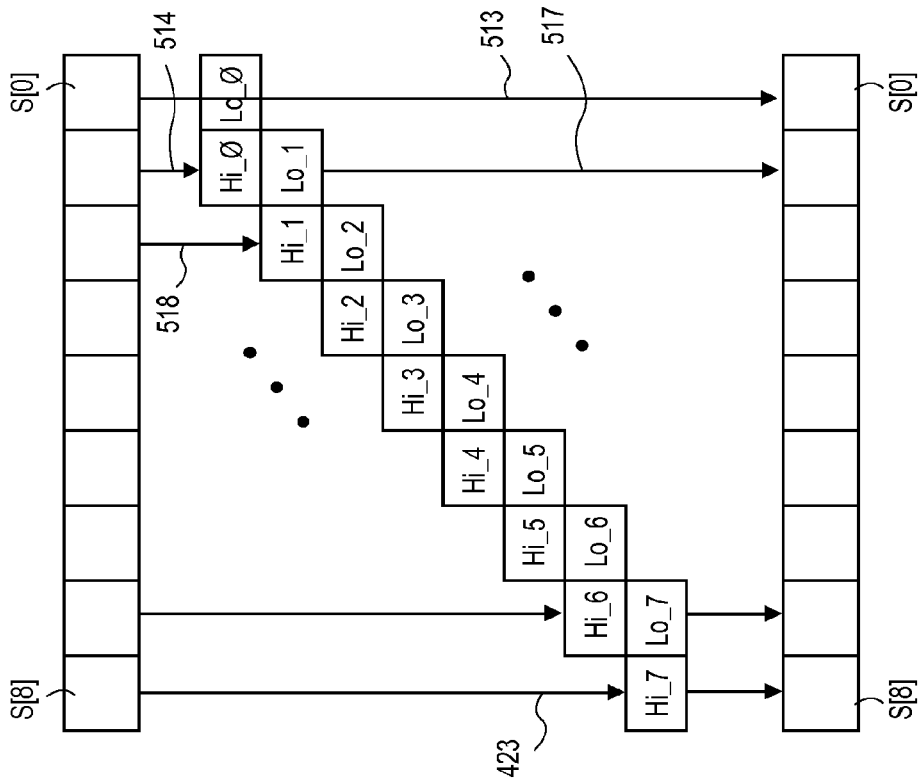


FIG. 5A

511 R_Lo ← VPMUL_LO A[0]:B[0]
512 R_Hi ← VPMUL_HI A[0]:B[0]
513 S[0] ← Add R_Lo; S[0]
514 S[1] ← Add R_Hi; S[1]
515 R_Lo ← VPMUL_LO A[0]:B[1]
516 R_Hi ← VPMUL_HI A[0]:B[1]
517 S[1] ← Add R_Lo; S[1]
518 S[2] ← Add R_Hi; S[2]

S[7] ← ADD R_LO; S[7]
S[8] ← ADD R_HI; S[8]

R_Lo ← VPMUL_LO A[0];B[j]

R_Hi ← VPMUL_HI A[0];B[j]

S[j];KØ ← ADD R_Lo; S[j]; KØ

S[j+1];K1 ← ADD R_Hi; S[j+1]; K1

FIG. 5B

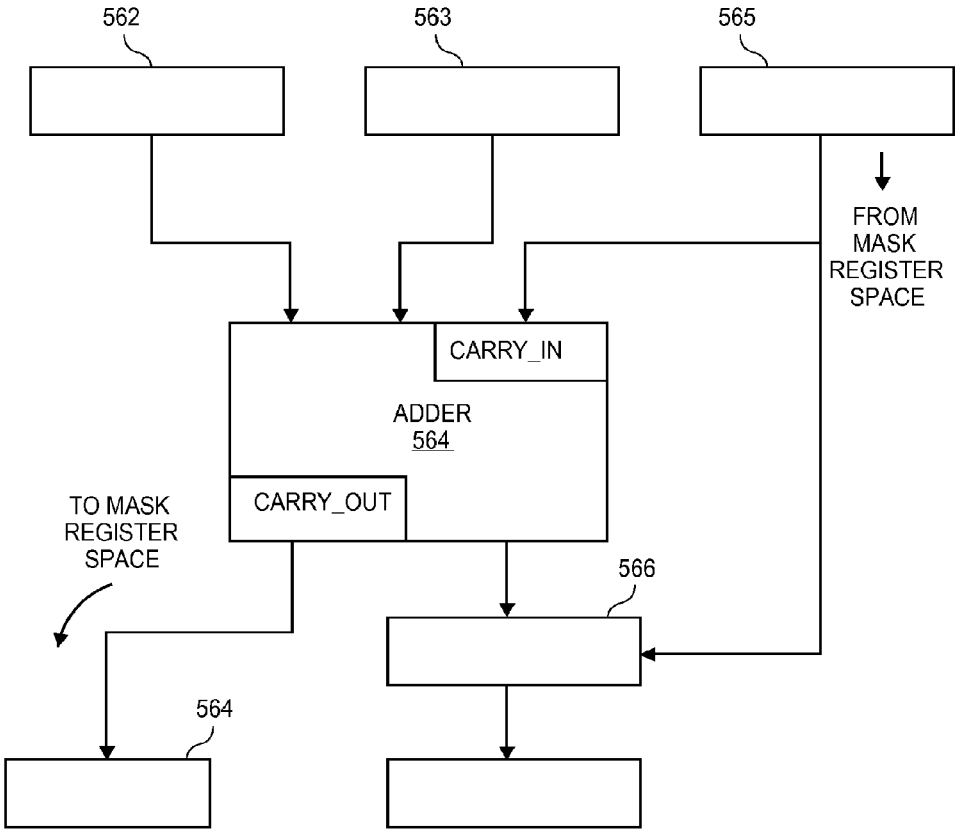

562
563
565
FROM MASK REGISTER SPACE
CARRY_IN
ADDER 564
CARRY_OUT
TO MASK REGISTER SPACE
566
564

FIG. 5C

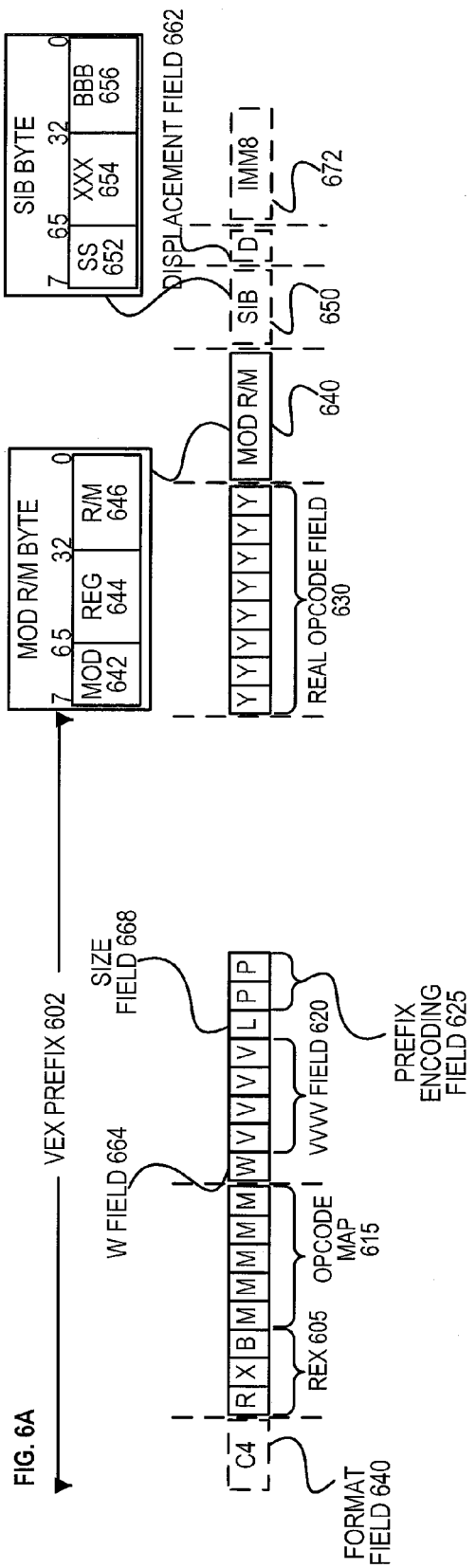


FIG. 6A
VEX PREFIX 602
SIZE FIELD 668
W FIELD 664
OPCODE MAP 615
VVVV FIELD 620
PREFIX ENCODING FIELD 625
REX 605
FORMAT FIELD 640
C4
R X B M M M M W V V V V L P P
REAL OPCODE FIELD 630
Y Y Y Y Y Y Y Y
MOD R/M BYTE
MOD 642
REG 644
R/M 646
MOD R/M 640
SIB BYTE
SS 652
XXX 654
BBB 656
SIB 650
DISPLACEMENT FIELD 662
D
IMM8 672

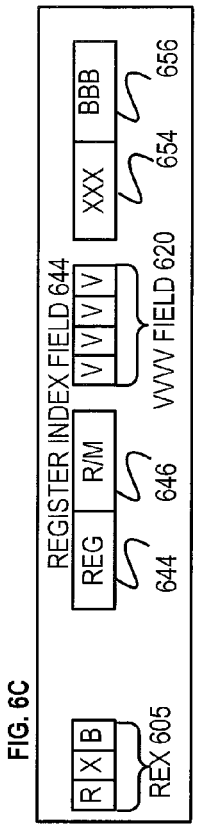


FIG. 6C
REGISTER INDEX FIELD 644
REG
R/M
644
646
V V V V
VVVV FIELD 620
XXX
BBB
654
656
R X B
REX 605

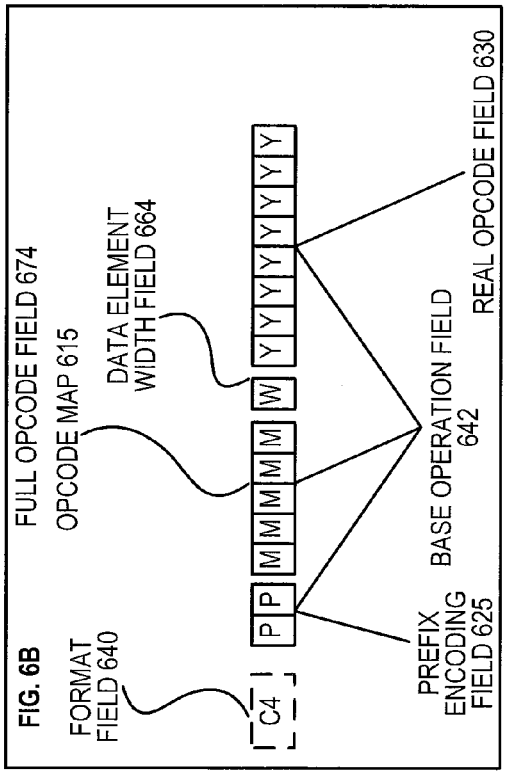


FIG. 6B
FULL OPCODE FIELD 674
OPCODE MAP 615
DATA ELEMENT WIDTH FIELD 664
FORMAT FIELD 640
C4
P P M M M M W Y Y Y Y Y Y Y Y
PREFIX ENCODING FIELD 625
BASE OPERATION FIELD 642
REAL OPCODE FIELD 630

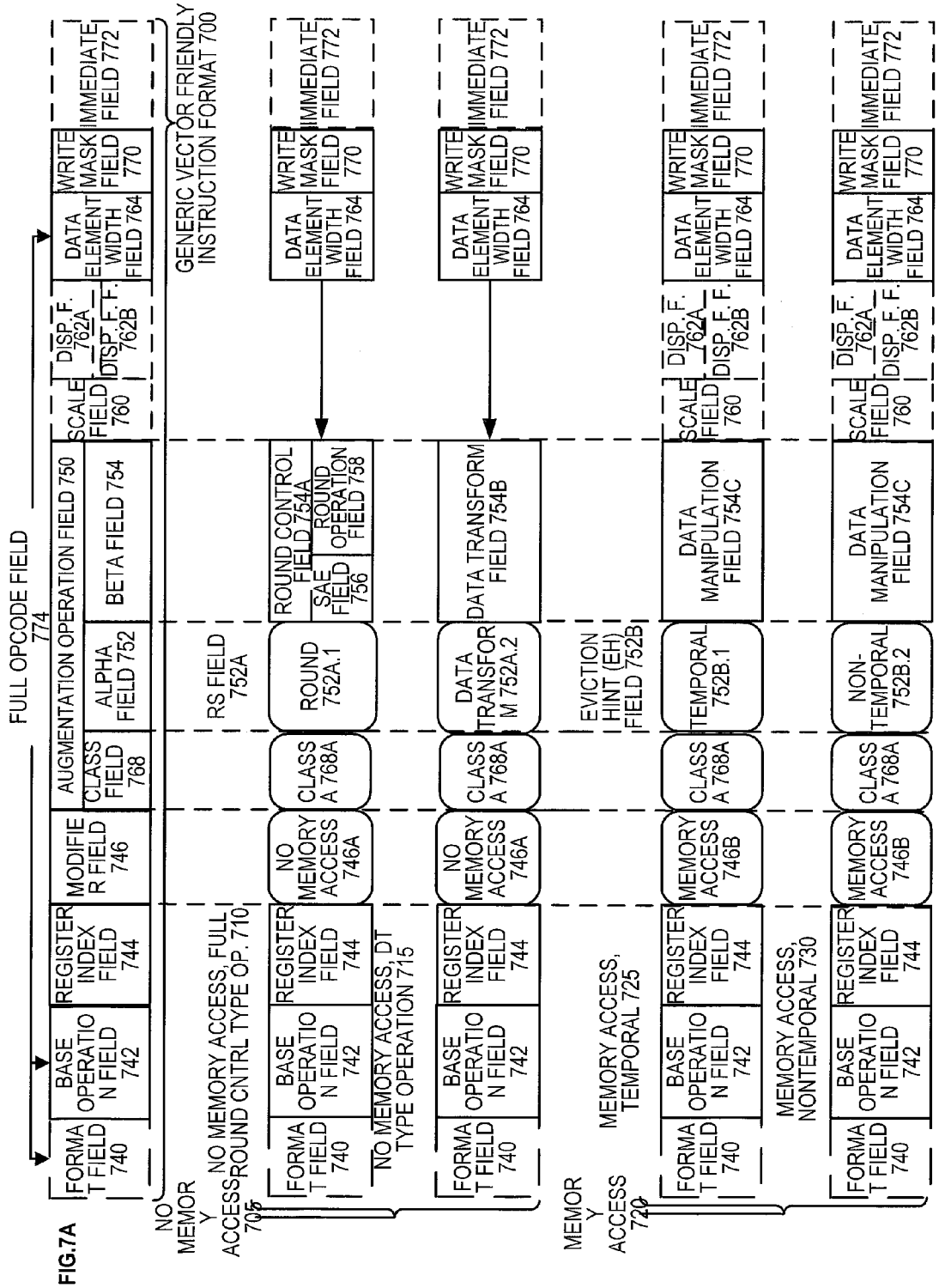


FIG.7A
GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 700
FULL OPCODE FIELD 774
NO MEMORY ACCESS 705
MEMORY ACCESS 720
FORMAT FIELD 740
BASE OPERATION FIELD 742
REGISTER INDEX FIELD 744
MODIFIER FIELD 746
AUGMENTATION OPERATION FIELD 750
CLASS FIELD 768
ALPHA FIELD 752
BETA FIELD 754
SCALE FIELD 760
DISP. F. 762A
DISP. F. F. 762B
DATA ELEMENT WIDTH FIELD 764
WRITE MASK FIELD 770
IMMEDIATE FIELD 772
NO MEMORY ACCESS, FULL ROUND CNTRL TYPE OP. 710
NO MEMORY ACCESS, DT TYPE OPERATION 715
MEMORY ACCESS, TEMPORAL 725
MEMORY ACCESS, NONTEMPORAL 730
NO MEMORY ACCESS 746A
MEMORY ACCESS 746B
CLASS A 768A
RS FIELD 752A
ROUND 752A.1
DATA TRANSFORM 752A.2
EVICTION HINT (EH) FIELD 752B
TEMPORAL 752B.1
NON-TEMPORAL 752B.2
ROUND CONTROL FIELD 754A
SAE FIELD 756
ROUND OPERATION FIELD 758
DATA TRANSFORM FIELD 754B
DATA MANIPULATION FIELD 754C

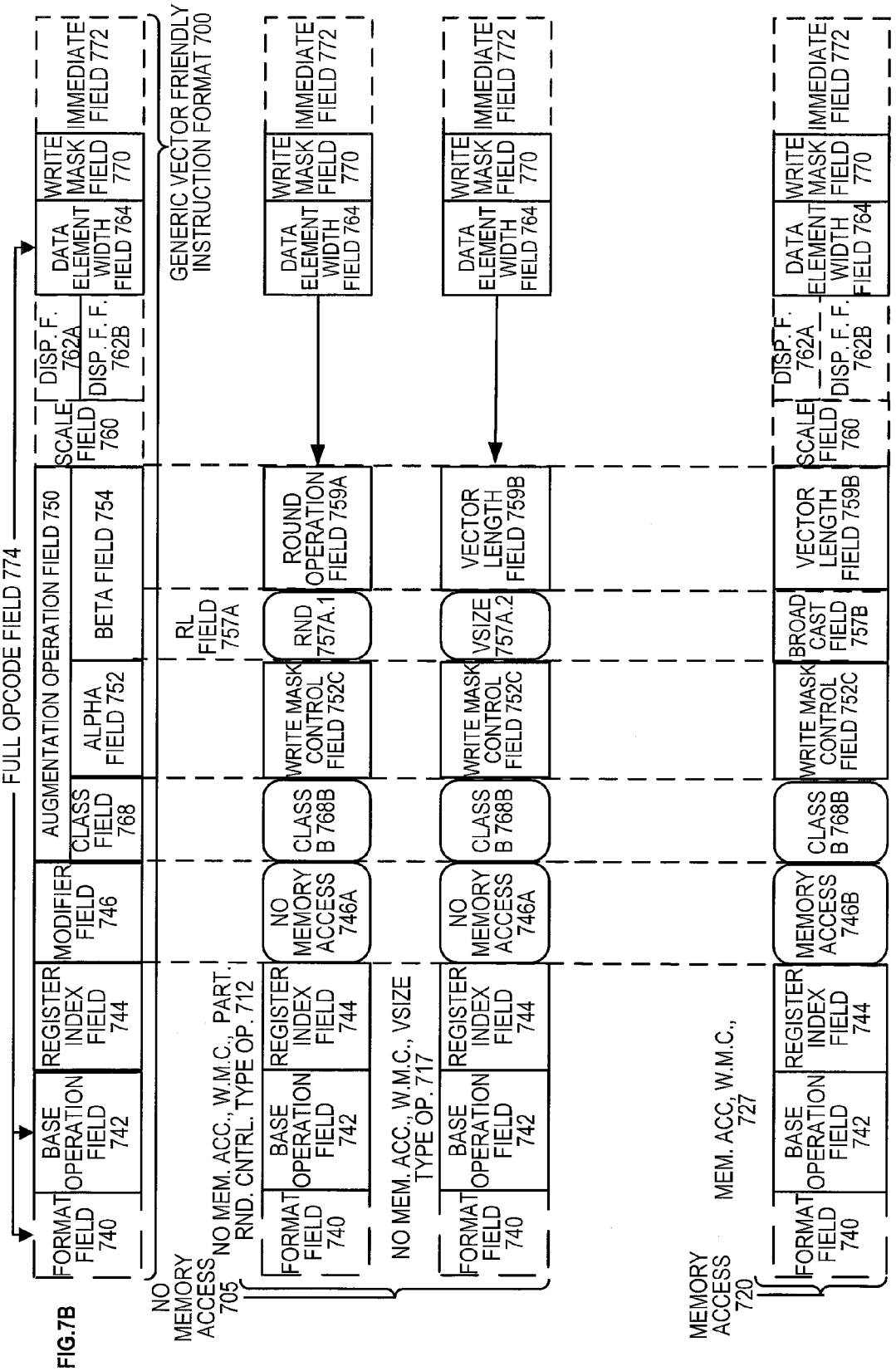

FULL OPCODE FIELD 774
GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 700
NO MEMORY ACCESS 705
NO MEM. ACC., W.M.C., PART. RND. CNTRL. TYPE OP. 712
NO MEM. ACC., W.M.C., VSIZE TYPE OP. 717
MEMORY ACCESS 720
MEM. ACC, W.M.C., 727
FORMAT FIELD 740
BASE OPERATION FIELD 742
REGISTER INDEX FIELD 744
MODIFIER FIELD 746
NO MEMORY ACCESS 746A
MEMORY ACCESS 746B
AUGMENTATION OPERATION FIELD 750
CLASS FIELD 768
CLASS B 768B
ALPHA FIELD 752
WRITE MASK CONTROL FIELD 752C
BETA FIELD 754
RL FIELD 757A
RND 757A.1
VSIZE 757A.2
ROUND OPERATION FIELD 759A
VECTOR LENGTH FIELD 759B
BROAD CAST FIELD 757B
SCALE FIELD 760
DISP. F. 762A
DISP. F. F. 762B
DATA ELEMENT WIDTH FIELD 764
WRITE MASK FIELD 770
IMMEDIATE FIELD 772

FIG.7B

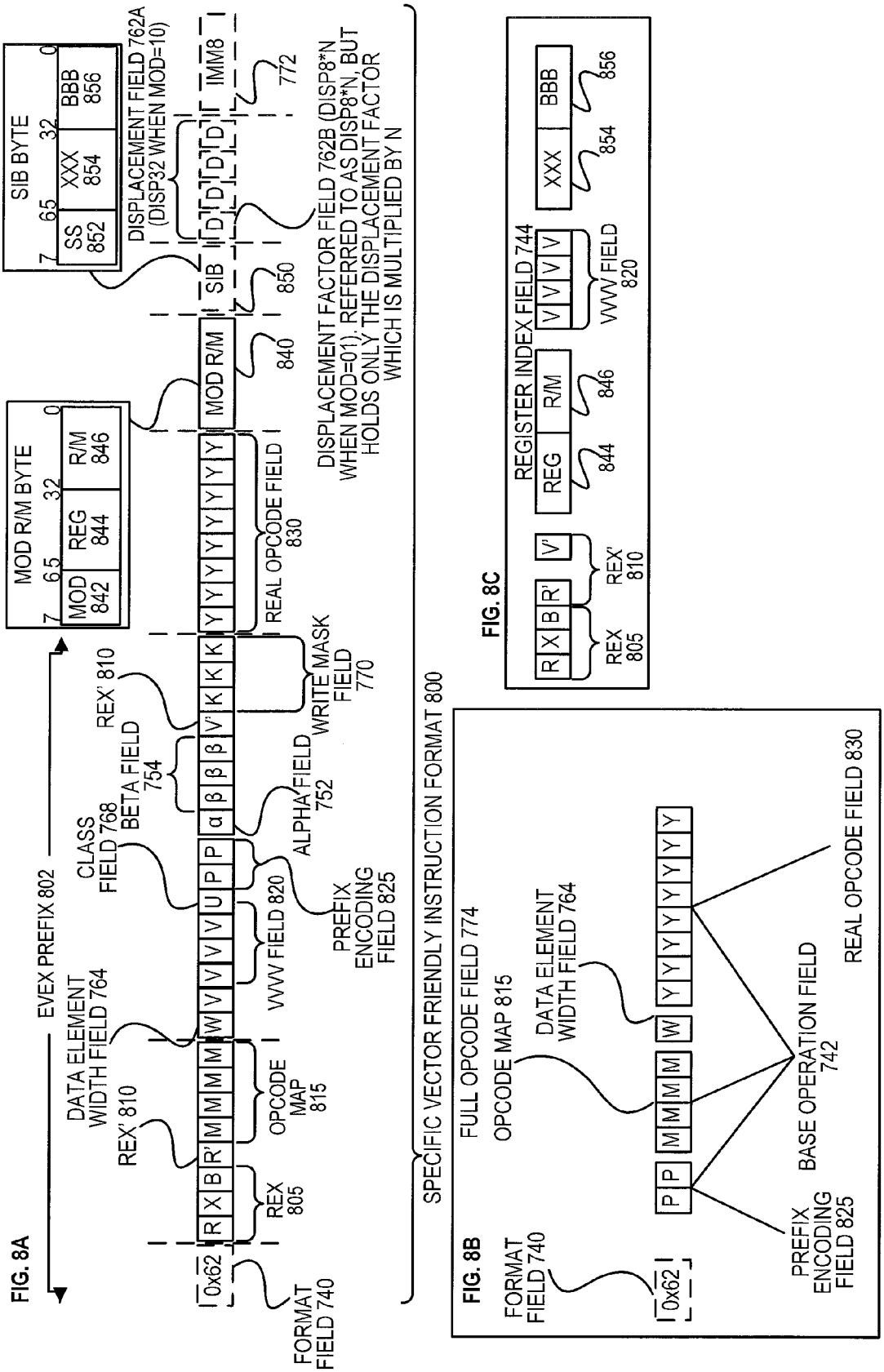
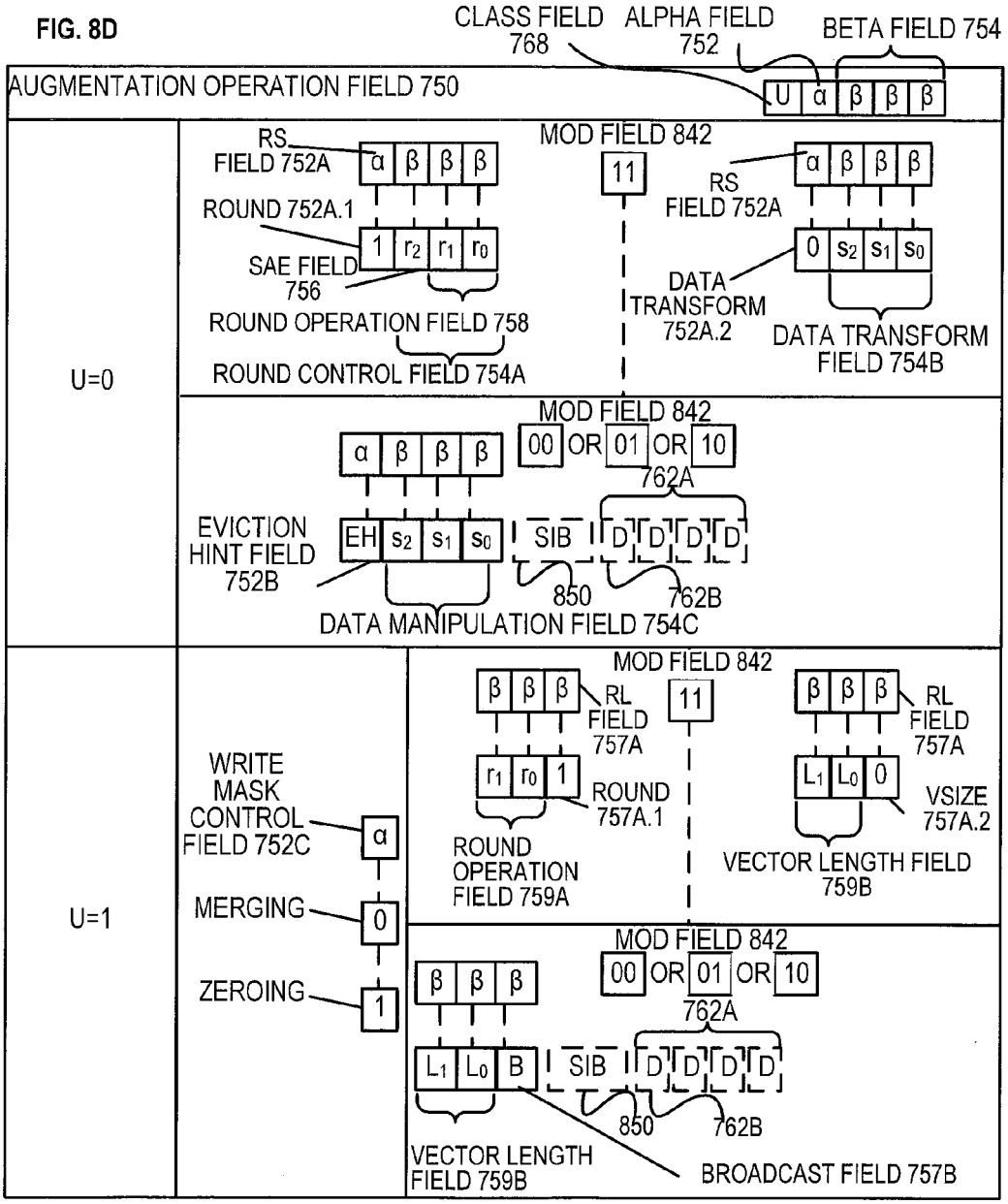


FIG. 8A
EVEX PREFIX 802
FORMAT FIELD 740
0x62
REX 805
REX' 810
OPCODE MAP 815
DATA ELEMENT WIDTH FIELD 764
VVVV FIELD 820
CLASS FIELD 768
BETA FIELD 754
ALPHA FIELD 752
PREFIX ENCODING FIELD 825
REX' 810
WRITE MASK FIELD 770
REAL OPCODE FIELD 830
MOD R/M BYTE
MOD 842
REG 844
R/M 846
MOD R/M 840
SIB BYTE
SS 852
XXX 854
BBB 856
SIB 850
DISPLACEMENT FIELD 762A (DISP32 WHEN MOD=10)
DISPLACEMENT FACTOR FIELD 762B (DISP8*N WHEN MOD=01), REFERRED TO AS DISP8*N, BUT HOLDS ONLY THE DISPLACEMENT FACTOR WHICH IS MULTIPLIED BY N
IMM8
772
SPECIFIC VECTOR FRIENDLY INSTRUCTION FORMAT 800
FIG. 8B
FULL OPCODE FIELD 774
FORMAT FIELD 740
0x62
PREFIX ENCODING FIELD 825
OPCODE MAP 815
DATA ELEMENT WIDTH FIELD 764
BASE OPERATION FIELD 742
REAL OPCODE FIELD 830
FIG. 8C
REGISTER INDEX FIELD 744
REG 844
R/M 846
REX 805
REX' 810
VVVV FIELD 820
XXX 854
BBB 856

FIG. 8D


CLASS FIELD 768
ALPHA FIELD 752
BETA FIELD 754
AUGMENTATION OPERATION FIELD 750
U α β β β
U=0
RS FIELD 752A
ROUND 752A.1
SAE FIELD 756
ROUND OPERATION FIELD 758
ROUND CONTROL FIELD 754A
MOD FIELD 842
11
RS FIELD 752A
DATA TRANSFORM 752A.2
DATA TRANSFORM FIELD 754B
MOD FIELD 842
00 OR 01 OR 10
762A
EVICTION HINT FIELD 752B
EH s2 s1 s0
SIB
D D D D
850
762B
DATA MANIPULATION FIELD 754C
U=1
WRITE MASK CONTROL FIELD 752C
MERGING
ZEROING
MOD FIELD 842
RL FIELD 757A
ROUND 757A.1
ROUND OPERATION FIELD 759A
VSIZE 757A.2
VECTOR LENGTH FIELD 759B
MOD FIELD 842
00 OR 01 OR 10
762A
L1 L0 B
SIB
D D D D
850
762B
VECTOR LENGTH FIELD 759B
BROADCAST FIELD 757B

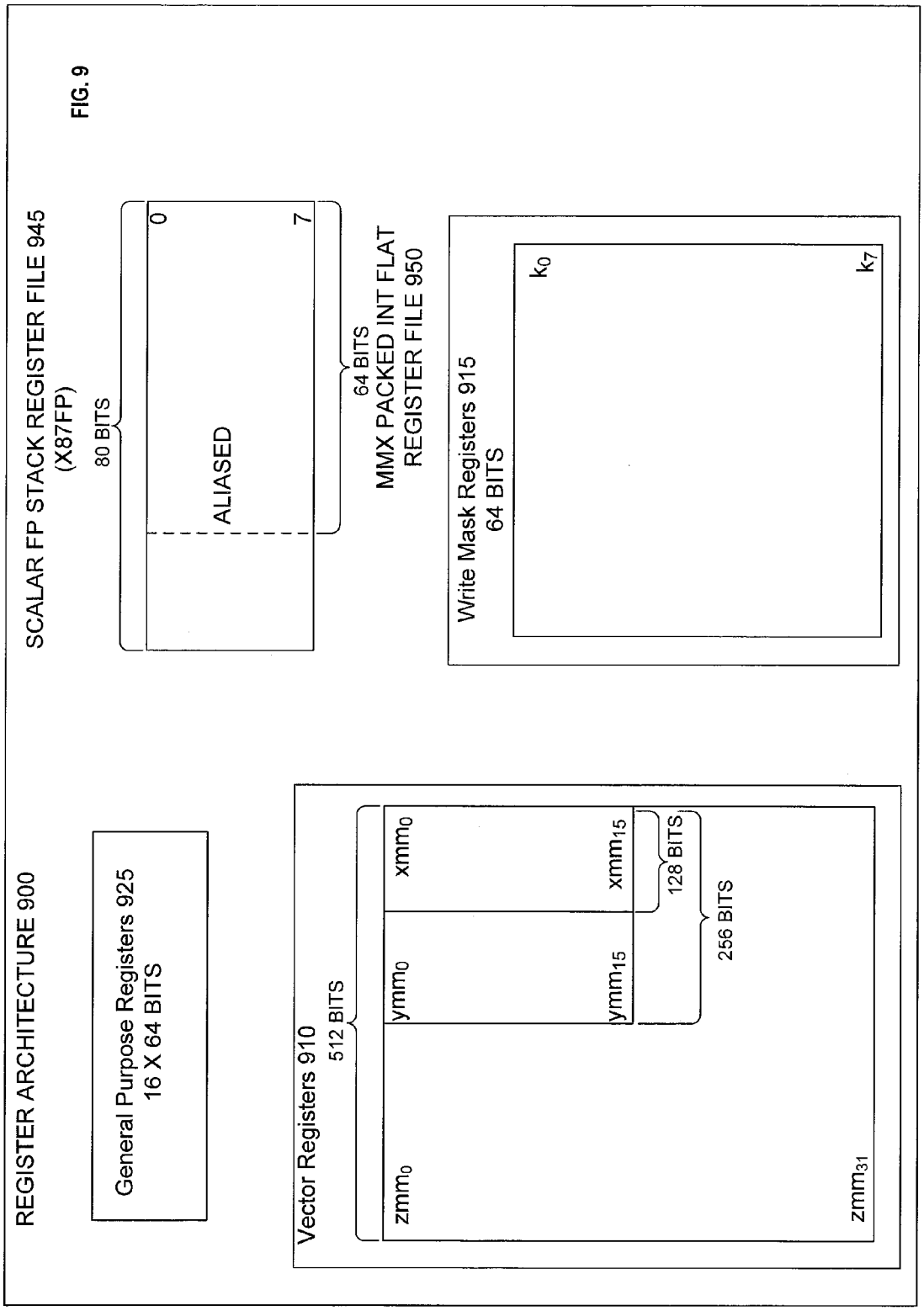

REGISTER ARCHITECTURE 900
General Purpose Registers 925
16 X 64 BITS
Vector Registers 910
512 BITS
zmm0
ymm0
xmm0
ymm15
xmm15
128 BITS
256 BITS
zmm31
SCALAR FP STACK REGISTER FILE 945
(X87FP)
80 BITS
0
7
ALIASED
64 BITS
MMX PACKED INT FLAT
REGISTER FILE 950
Write Mask Registers 915
64 BITS
k0
k7

FIG. 9

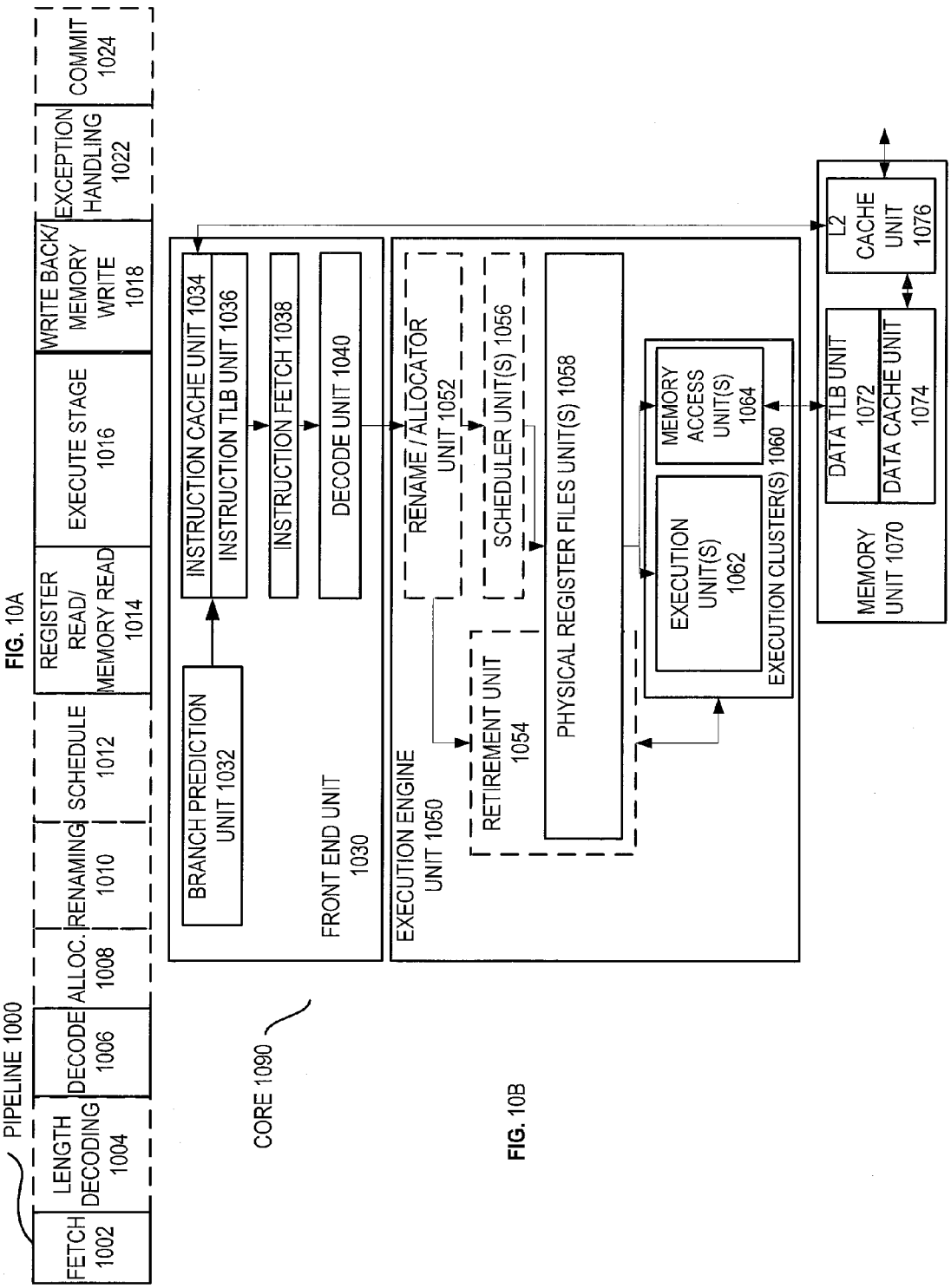

PIPELINE 1000
FETCH 1002
LENGTH DECODING 1004
DECODE 1006
ALLOC. 1008
RENAMING 1010
SCHEDULE 1012
REGISTER READ/ MEMORY READ 1014
EXECUTE STAGE 1016
WRITE BACK/ MEMORY WRITE 1018
EXCEPTION HANDLING 1022
COMMIT 1024
FIG. 10A
CORE 1090
FRONT END UNIT 1030
BRANCH PREDICTION UNIT 1032
INSTRUCTION CACHE UNIT 1034
INSTRUCTION TLB UNIT 1036
INSTRUCTION FETCH 1038
DECODE UNIT 1040
EXECUTION ENGINE UNIT 1050
RENAME / ALLOCATOR UNIT 1052
RETIREMENT UNIT 1054
SCHEDULER UNIT(S) 1056
PHYSICAL REGISTER FILES UNIT(S) 1058
EXECUTION UNIT(S) 1062
MEMORY ACCESS UNIT(S) 1064
EXECUTION CLUSTER(S) 1060
MEMORY UNIT 1070
DATA TLB UNIT 1072
DATA CACHE UNIT 1074
L2 CACHE UNIT 1076
FIG. 10B

FIG. 11B

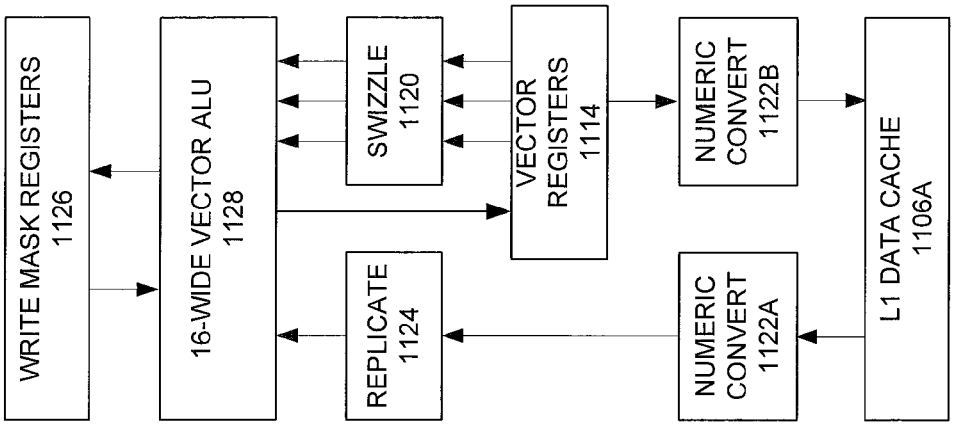
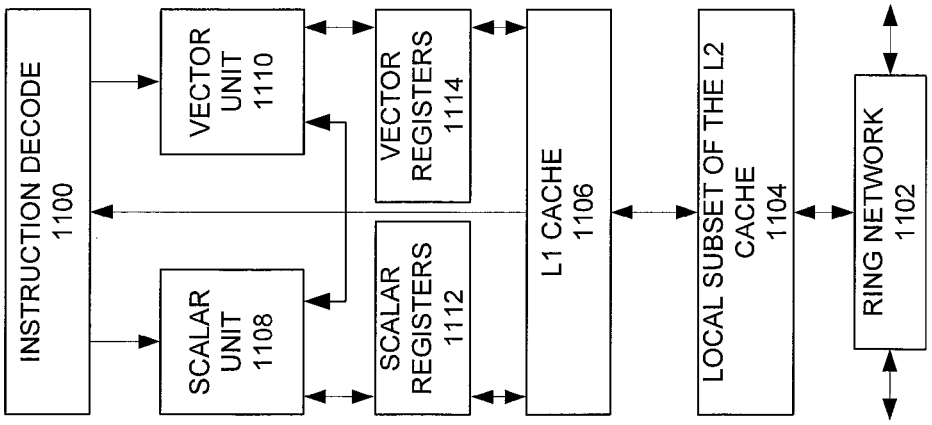


FIG. 11A



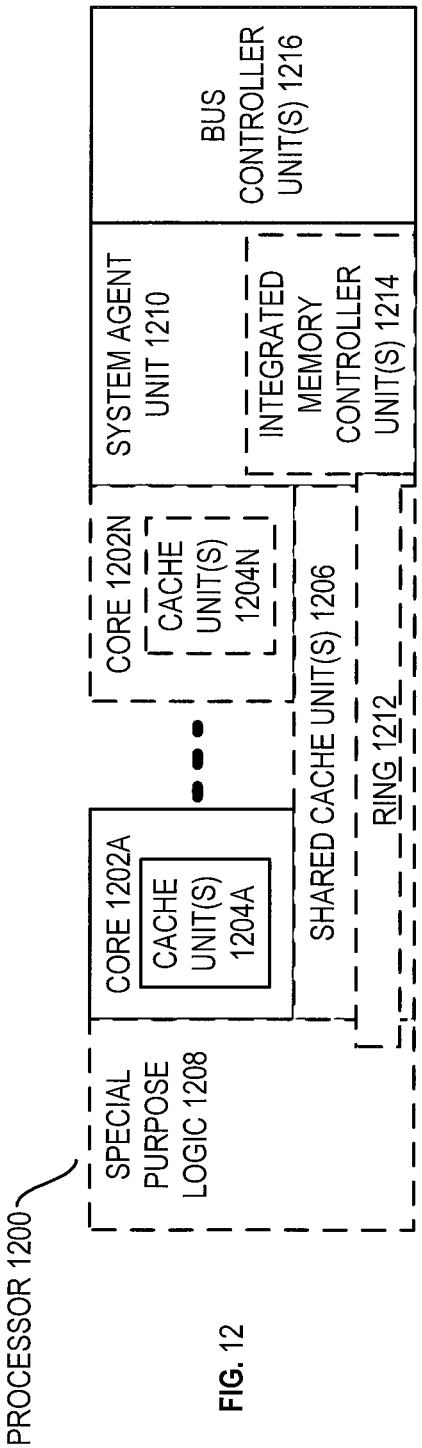

PROCESSOR 1200
SPECIAL PURPOSE LOGIC 1208
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SHARED CACHE UNIT(S) 1206
RING 1212
SYSTEM AGENT UNIT 1210
INTEGRATED MEMORY CONTROLLER UNIT(S) 1214
BUS CONTROLLER UNIT(S) 1216

FIG. 12

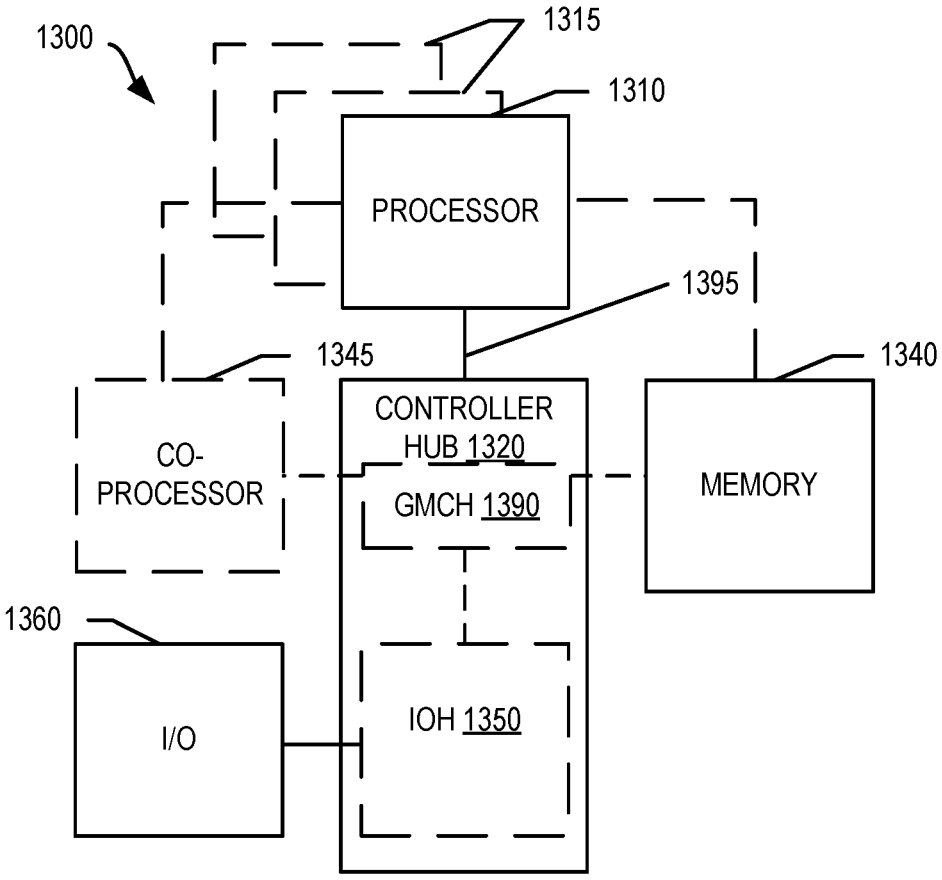

1300
1315
1310
PROCESSOR
1395
1345
1340
CONTROLLER
HUB 1320
CO-
PROCESSOR
GMCH 1390
MEMORY
1360
I/O
IOH 1350

FIG. 13

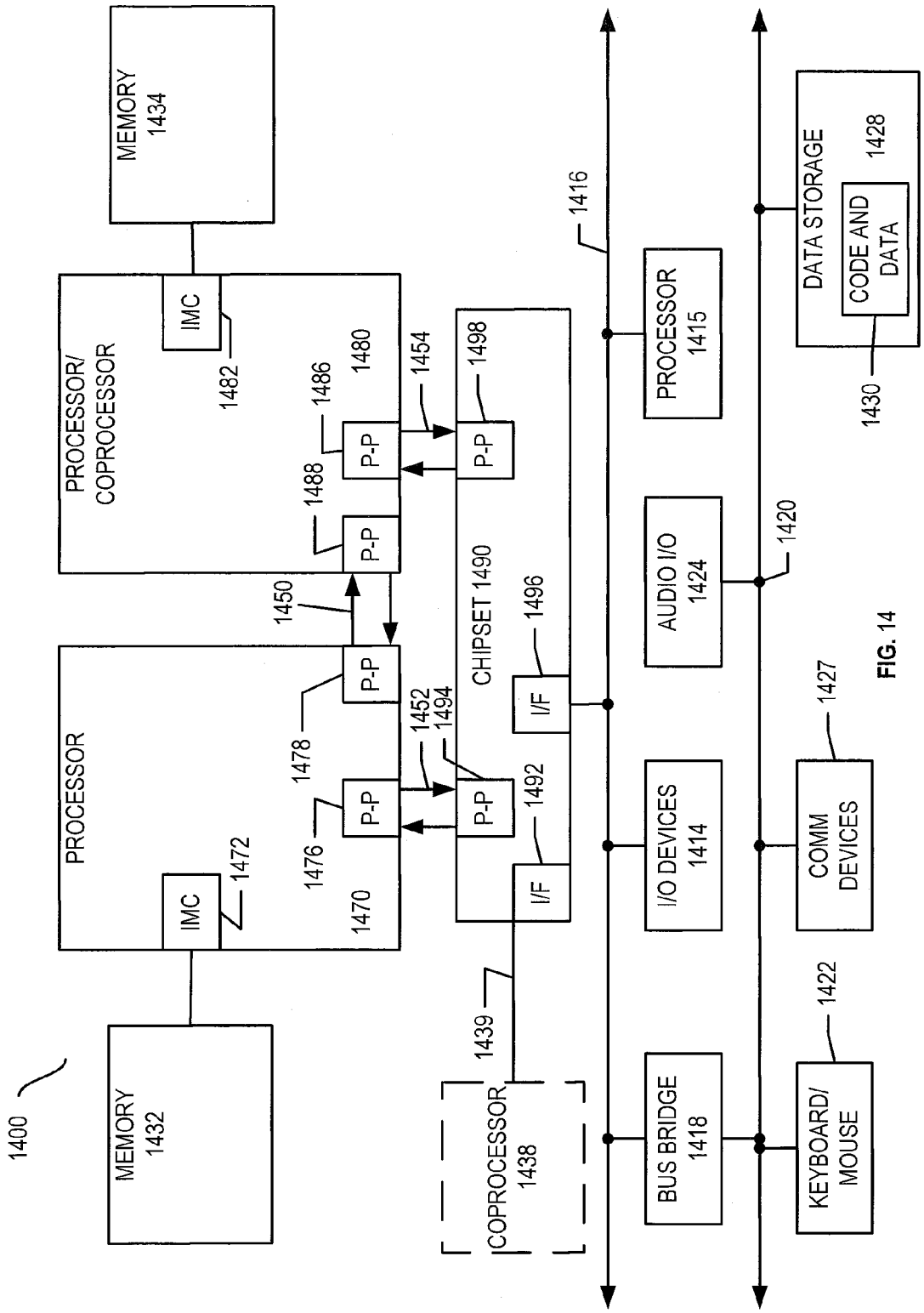

1400
MEMORY 1432
PROCESSOR
IMC
1472
1470
1476
1478
P-P
P-P
1450
1452
1494
PROCESSOR/ COPROCESSOR
IMC
1482
1480
1486
1488
P-P
P-P
1454
1498
MEMORY 1434
CHIPSET 1490
I/F
1496
I/F
1492
1439
COPROCESSOR 1438
1416
PROCESSOR 1415
AUDIO I/O 1424
I/O DEVICES 1414
BUS BRIDGE 1418
1420
DATA STORAGE
CODE AND DATA
1428
1430
1427
COMM DEVICES
1422
KEYBOARD/ MOUSE

FIG. 14

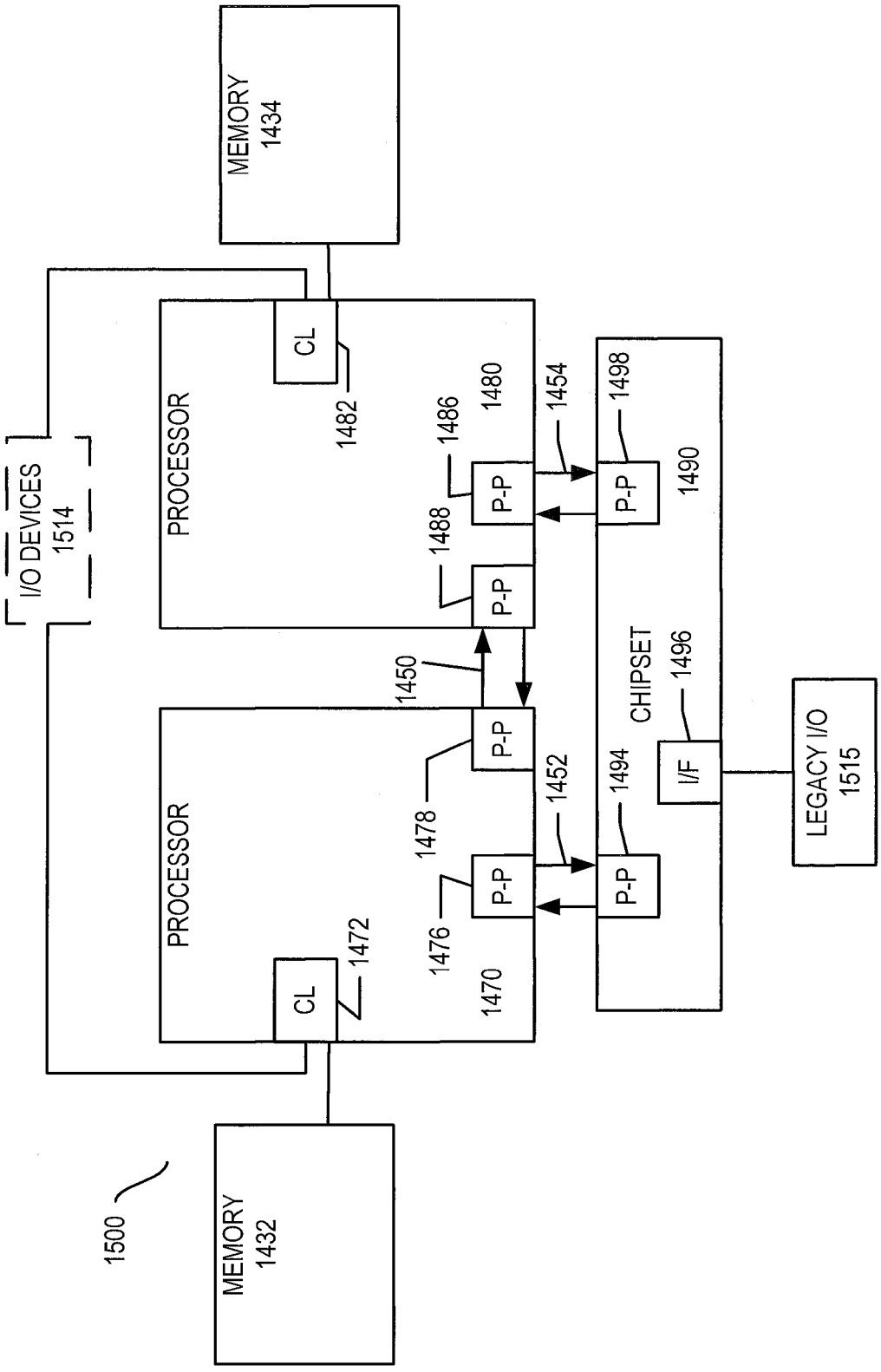

MEMORY
1434
I/O DEVICES
1514
PROCESSOR
CL
1482
1480
1486
1488
P-P
P-P
1454
1498
P-P
1490
1450
PROCESSOR
P-P
1478
1452
P-P
1476
1470
CL
1472
CHIPSET
1496
I/F
1494
P-P
LEGACY I/O
1515
1500
MEMORY
1432

FIG. 15

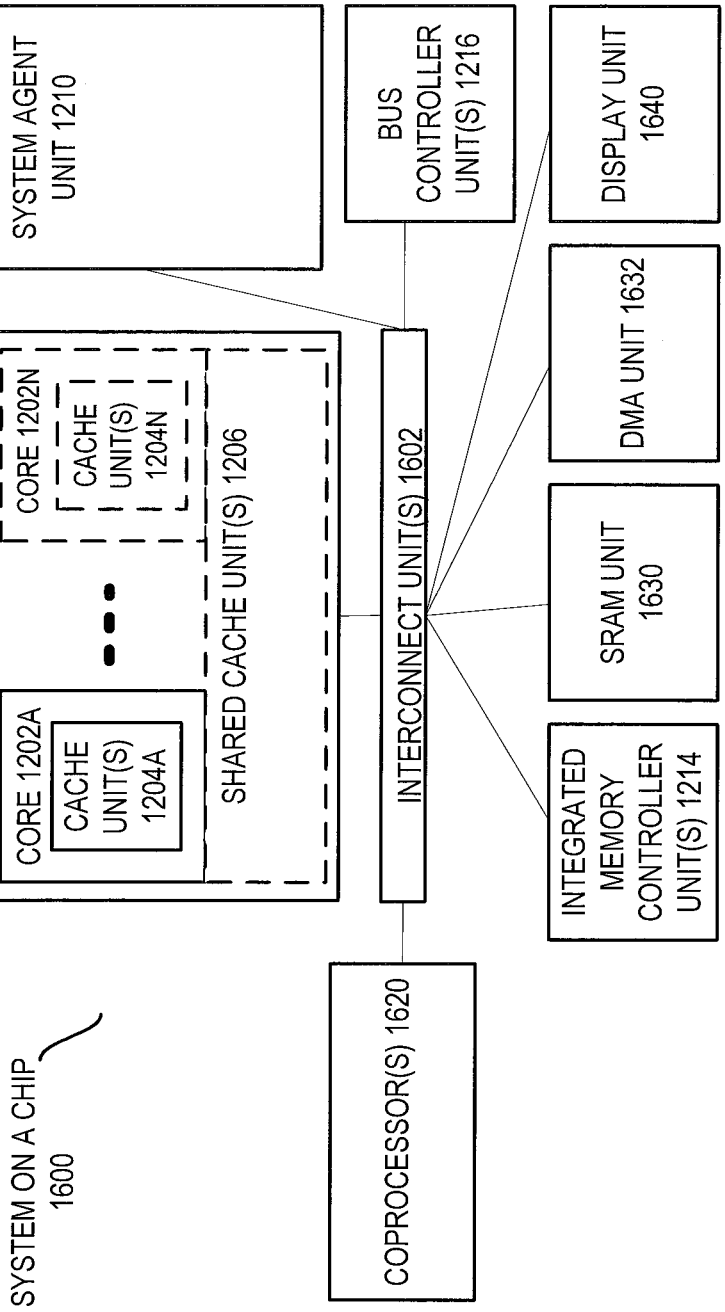

SYSTEM ON A CHIP
1600
APPLICATION PROCESSOR 1610
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SHARED CACHE UNIT(S) 1206
SYSTEM AGENT UNIT 1210
INTERCONNECT UNIT(S) 1602
BUS CONTROLLER UNIT(S) 1216
DISPLAY UNIT 1640
DMA UNIT 1632
SRAM UNIT 1630
INTEGRATED MEMORY CONTROLLER UNIT(S) 1214
COPROCESSOR(S) 1620

FIG. 16

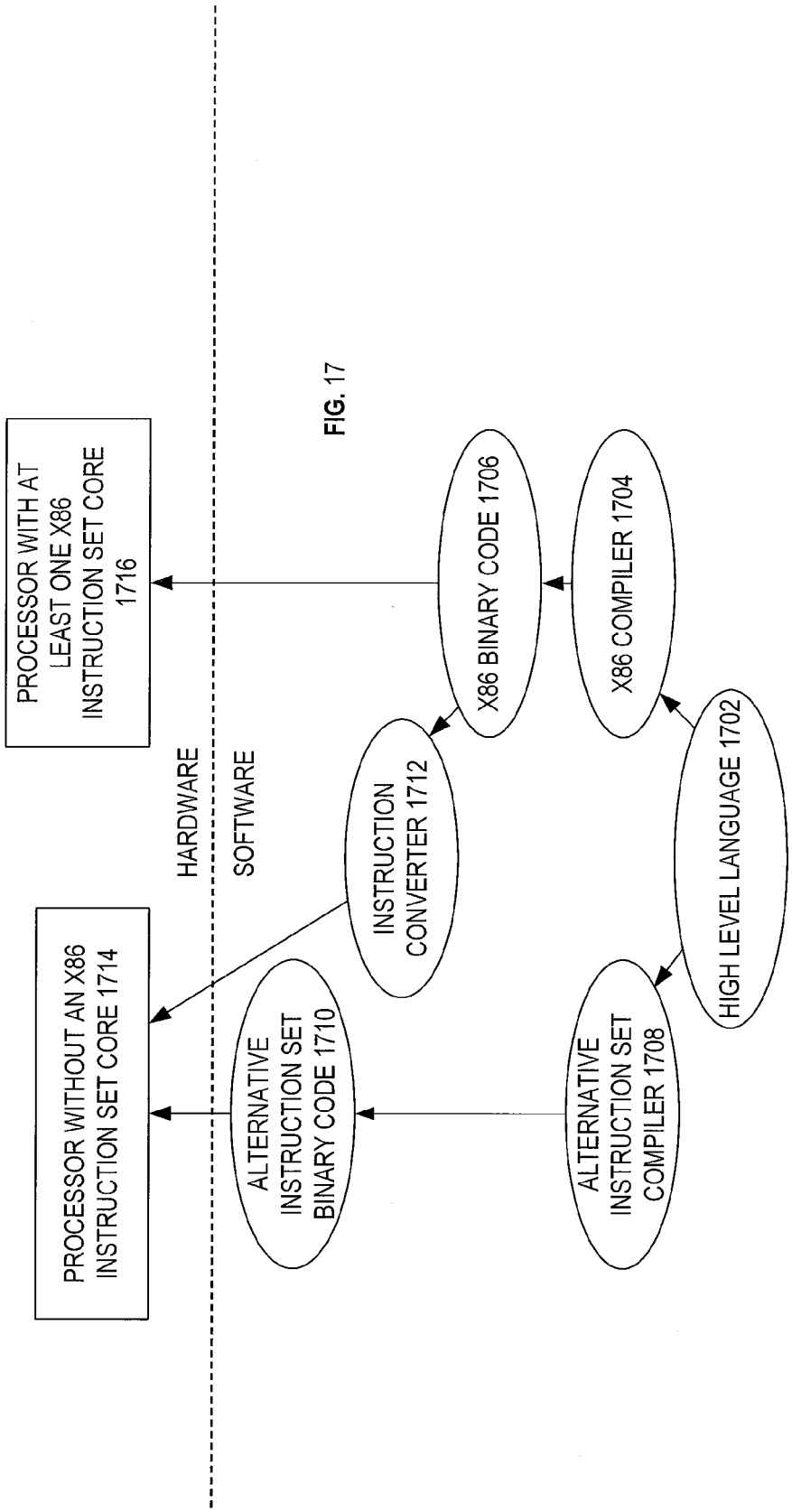

PROCESSOR WITH AT LEAST ONE X86 INSTRUCTION SET CORE 1716
HARDWARE
SOFTWARE
X86 BINARY CODE 1706
X86 COMPILER 1704
INSTRUCTION CONVERTER 1712
HIGH LEVEL LANGUAGE 1702
PROCESSOR WITHOUT AN X86 INSTRUCTION SET CORE 1714
ALTERNATIVE INSTRUCTION SET BINARY CODE 1710
ALTERNATIVE INSTRUCTION SET COMPILER 1708

FIG. 17

APPARATUS AND METHOD FOR VECTOR INSTRUCTIONS FOR LARGE INTEGER ARITHMETIC

CROSS-REFERENCE TO RELATED APPLICATION

This patent application is a U.S. National Phase Application under 35 U.S.C. §371 of International Application No. PCT/US2011/067165, filed Dec. 23, 2011, entitled APPARATUS AND METHOD FOR VECTOR INSTRUCTIONS FOR LARGE INTEGER ARITHMETIC.

BACKGROUND

1. Field of Invention

The present invention pertains to the computing sciences generally, and, more specifically to an apparatus and method for vector instructions for large integer arithmetic.

2. Background

FIG. **1** shows a high level diagram of a processing core **100** implemented with logic circuitry on a semiconductor chip. The processing core includes a pipeline **101**. The pipeline consists of multiple stages each designed to perform a specific step in the multi-step process needed to fully execute a program code instruction. These typically include at least: 1) instruction fetch and decode; 2) data fetch; 3) execution; 4) write-back. The execution stage performs a specific operation identified by an instruction that was fetched and decoded in prior stage(s) (e.g., in step 1) above) upon data identified by the same instruction and fetched in another prior stage (e.g., step 2) above). The data that is operated upon is typically fetched from (general purpose) register storage space **102**. New data that is created at the completion of the operation is also typically "written back" to register storage space (e.g., at stage 4) above).

The logic circuitry associated with the execution stage is typically composed of multiple "execution units" or "functional units" **103_1** to **103**_N that are each designed to perform its own unique subset of operations (e.g., a first functional unit performs integer math operations, a second functional unit performs floating point instructions, a third functional unit performs load/store operations from/to cache/memory, etc.). The collection of all operations performed by all the functional units corresponds to the "instruction set" supported by the processing core **100**.

Two types of processor architectures are widely recognized in the field of computer science: "scalar" and "vector". A scalar processor is designed to execute instructions that perform operations on a single set of data, whereas, a vector processor is designed to execute instructions that perform operations on multiple sets of data. FIGS. **2**A and **2**B present a comparative example that demonstrates the basic difference between a scalar processor and a vector processor.

FIG. **2**A shows an example of a scalar AND instruction in which a single operand set, A and B, are ANDed together to produce a singular (or "scalar") result C (i.e., AB=C). By contrast, FIG. **2**B shows an example of a vector AND instruction in which two operand sets, A/B and D/E, are respectively ANDed together in parallel to simultaneously produce a vector result C, F (i.e., A.AND.B=C and D.AND.E=F). As a matter of terminology, a "vector" is a data element having multiple "elements". For example, a vector V=Q, R, S, T, U has five different elements: Q, R, S, T and U. The "size" of the exemplary vector V is five (because it has five elements).

FIG. **1** also shows the presence of vector register space **104** that is different that general purpose register space **102**. Specifically, general purpose register space **102** is nominally used to store scalar values. As such, when, the any of execution units perform scalar operations they nominally use operands called from (and write results back to) general purpose register storage space **102**. By contrast, when any of the execution units perform vector operations they nominally use operands called from (and write results back to) vector register space **107**. Different regions of memory may likewise be allocated for the storage of scalar values and vector values.

Note also the presence of masking logic **104_1** to **104**_N and **105_1** to **105**_N at the respective inputs to and outputs from the functional units **103_1** to **103**_N. In various implementations, only one of these layers is actually implemented—although that is not a strict requirement. For any instruction that employs masking, input masking logic **104_1** to **104**_N and/or output masking logic **105_1** to **105**_N may be used to control which elements are effectively operated on for the vector instruction. Here, a mask vector is read from a mask register space **106** (e.g., along with input data vectors read from vector register storage space **107**) and is presented to at least one of the masking logic **104**, **105** layers.

Over the course of executing vector program code each vector instruction need not require a full data word. For example, the input vectors for some instructions may only be 8 elements, the input vectors for other instructions may be 16 elements, the input vectors for other instructions may be 32 elements, etc. Masking layers **104/105** are therefore used to identify a set of elements of a full vector data word that apply for a particular instruction so as to effect different vector sizes across instructions. Typically, for each vector instruction, a specific mask pattern kept in mask register space **106** is called out by the instruction, fetched from mask register space and provided to either or both of the mask layers **104/105** to "enable" the correct set of elements for the particular vector operation.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings, in which like references indicate similar elements and in which:

FIG. **1** shows an instruction execution pipeline;

FIGS. **2***a* and **2***b* compare scalar vs. vector processing;

FIG. **3***a* through FIG. **3***c* show a mathematical perspective for the multiplication of two large numbers;

FIGS. **4***a* through **4***d* pertain to a first embodiment of an instruction set and sequence code thereof for the multiplication of two large numbers;

FIGS. **5***a* through **5***c* pertain to a first embodiment of an instruction set and sequence code thereof for the multiplication of two large numbers;

FIG. **6**A illustrates an exemplary AVX instruction format;

FIG. **6**B illustrates which fields from FIG. **6**A make up a full opcode field and a base operation field;

FIG. **6**C illustrates which fields from FIG. **6**A make up a register index field;

FIGS. **7**A-**7**B are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention;

FIG. **8** is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention;

FIG. **9** is a block diagram of a register architecture according to one embodiment of the invention;

FIG. **10**A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention;

FIG. **10**B is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention;

FIGS. **11**A-B illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip;

FIG. **12** is a block diagram of a processor that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention;

FIG. **13** is a block diagram of a exemplary system in accordance with an embodiment of the present invention;

FIG. **14** is a block diagram of a first more specific exemplary system in accordance with an embodiment of the present invention;

FIG. **15** is a block diagram of a second more specific exemplary system in accordance with an embodiment of the present invention;

FIG. **16** is a block diagram of a SoC in accordance with an embodiment of the present invention;

FIG. **17** is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention.

DETAILED DESCRIPTION

Overview

Detailed Description

FIG. **3***a* through FIG. **3***c* show a mathematical perspective for the multiplication of two large numbers that forms the basis for vector integer instructions described in more detail further below. For simplicity, the integers being multiplied in FIG. **3***a* are not very large and, moreover, are expressed in base 10 form (ten possible digits 0 through 9) rather than in base 2 form (two possible digits 0 and 1). Nevertheless, they are sufficient to bring forward pertinent aspects of the instructions described herein which are capable of multiplying much larger numbers expressed in base 2 form.

As observed in FIG. **3***a*, a multiplicand B=765 is multiplied **301** by a multiplier A=834. The summation of partial products **302** is consistent with elementary mathematics and shows the final result to be 638,010. Notably, the three partial products **302***a*, **302***b*, **302***b* can be viewed akin to a "right-wise staircase" structure **303**, where: 1) the lowest ordered partial product **302***a* corresponds to the multiplication of the lowest ordered digit of the multiplier A[0]=4 by all three digits of the multiplicand B[2:0]=765; 2) the middle ordered partial product **302***b* is shifted to the left one place relative to the lowest ordered partial product **302***a* and corresponds to the multiplication of the middle ordered digit of the multiplier A[1]=3 by all three digits of the multiplicand B[2:0]=765; and, 3) the highest ordered partial product **302***c* is shifted to the left one place relative to the middle ordered partial product **302***b* and corresponds to the multiplication of the highest ordered digit of the multiplier A[2]=8 by all three digits of the multiplicand B[2:0]=765.

As such, the three partial products can be expressed as: 1) A[0]*B[2:0] for the lowest ordered partial product **302***a;* 2) A[1]*B[2:0] for the middle ordered partial product **302***b*; and, 3) A[2]*B[2:0] for the highest ordered partial product **302***c*.

FIG. **3***b* shows a perspective for the determination of the partial products. Specifically, the calculation of each of the partial products **302***a,b,c* can also be viewed as a respective right-wise staircase structure much the same as discussed just above. For example, the lowest ordered partial product **302***a* can be determined by summing over three sub-partial products **304***a,b,c*. Here, the first sub-partial product **304***a* corresponds to A[0]*B[0] (i.e., 4*5=20), the second sub-partial product **304***b* corresponds to A[0]*B[1] (i.e., 4*6=24) shifted left one digit location relative the first sub-partial product **304***a*, and, the third sub-partial product **304***c* corresponds to A[0]*B[2] (i.e., 4*7=28) shifted left one digit location relative to the second sub-partial product **304***b*.

The partial product **302***a* is determined by adding the sub-partial products consistent with their alignment as indicated by arrows **305***a-d*. Note that carry terms are respected as indicated by carry term **306**. The remaining partial products **302***b* and **302***c* are determined in like manner as observed in insets **307** and **308**.

FIG. **3***c* shows a flow diagram that illustrates a method of multiplication consistent with some of the principles discussed above. Partial product digits are accumulated in a storage element S **320**. For the recursion **330** of the first partial product term, the storage element S is initialized with a value of 0 for all digits **320_1**. A first partial product is determined by selecting a lowest ordered digit in a multiplier (A[0]) and multiplying **310** it with the lowest ordered digit in a multiplicand (B[0]). The lowest ordered digit in the multiplier A[0] is then multiplied **311** against the next higher ordered multiplicand (B[1]). The least significant digit of the two sub partial products is added with its corresponding (aligned) digit in storage element S **320_1** and re-stored in storage element S **320_2**. The pair of digits of the two sub-partial products having overlapping alignment are added 313 with their corresponding (aligned) digit of storage element S **320_1**. The results of addition **313** are kept in storage element **320_2**.

The lowest ordered digit in the multiplier A[0] is next multiplied **314** against the next higher digit in the multiplicand **314** (B[2]) and the result is added 315 with the highest ordered bit of sub partial product **311** and their corresponding (aligned) digit in storage element S. The result of addition **315** is re-stored in storage element S **320_2**. Note that a carry term is generated **316** with addition **315**.

Because the B[2] term is the highest ordered digit in the multiplicand, the highest ordered digit of sub-partial product **314** is added 317 to its corresponding (aligned) digit in storage element S and the carry term. At this point, the first partial product is stored in storage element S **320_2**. Those of ordinary skill will appreciate that various "kernels" of multiplication, alignment, addition and storage processes can be devised that are repeated for multiple additional digit locations depending on the size of the multiplicand.

With the first partial product being stored in storage element **320_2**, a substantially similar process as that of process **330** is used to calculate the second partial product A[1]*B[2:0] with the resulting accumulation of partial products being left in storage element S **320_3**. As with the calculation of the first partial product, for each digit in the multiplicand B, there is a multiplication with the multiplier

term (A[1] in this case), the resultant is properly aligned and aligned digits of two consecutive products are added. An additional feature of the calculation of the second partial product is that its "rightwise staircase" structure is aligned one digit to the left relative to the "rightwise staircase" structure of the previous (first) partial product.

The third partial product is calculated with the same approach and the final result of the multiplication is stored in storage element **320_4**. Those of ordinary skill will recognize that, although only three iterations are shown (because the multiplicand only has three digits), the recursion described above can be extended to include more or less iterations depending on the size of the multiplicand.

FIGS. **4***a,b* and **5***a,b* pertain to an instruction set and variations thereof for implementation in a semiconductor processing unit (e.g. a processing core of a multi-core CPU). Here, two large integer values A and B are to be multiplied. In an embodiment, A and B can each be as large as 512 bits. In a further embodiment, each "digit" of A and B is viewed as a 64 bit value within the overall 512 bit structure. As such, each of A and B can be viewed as being as large as an 8 element vector where each element in the vector represents a digit, and, each digit is 64 bits.

According to this perspective, the partial product recursions take the form of A[i]*B[7:0] where A[i] represents a particular digit in the multiplicand A and B[7:0] represents each digit in the multiplier B. As described in further detail below, similar to the approach just discussed above, the multiplication of A*B is performed by determining the partial product A[i]*B[7:0] for each value of i where i represents a different digit in the multiplicand A. Also similar to the approach just discussed above, aligned digits of a same partial product recursion are added together along with a value along the same alignment position that was stored from the previously calculated partial product recursion. These and other features will be more apparent through discussion of the immediately following example.

FIG. **4***a* shows an instruction sequence **401** that calculates a partial product for the A[0] multiplier term. Here, the instruction sequence can be viewed as calculating the product of A[0]*B[j] for each of j recursions where j=0 to 7 (for a maximum sized multiplicand B). Because both the A[0] and the B[j] term corresponds to a 64 bit digit, 128 bits are allocated for the product of the two. FIG. **4***a* shows the right-wise stair case structure effected by the instruction sequence. Each sub-partial product is represented by a 128 bit data structure consisting of a 64 bit lower half ("Lo") and a 64 bit upper half ("Hi").

The instruction sequence **401** relies on a class of multiplication instructions that return the low half or upper half of a sub partial product A[i] *B [j] term. A first instruction **411** VPMUL_LO calculates the first sub partial product term (A[0] *B [0]) and returns its lower half (Lo_0) in resultant register R_Lo. Partial product terms, as opposed to sub-partial product terms, are accumulated in register S. Here, S is a vector where each element in vector S corresponds to a 64 bit digit in the accumulated partial product value contained in vector S. Instruction sequence **401** corresponds to the initial recursion (i.e., the recursion for the A[0] term), thus vector S is initialized beforehand with a value 0 for all digits.

The second instruction **412** performs aligned addition by adding the contents of R_Lo with the lowest ordered element/digit in S (S[0]=0) and re-storing it in S. Instructions **411** and **412** serve as an initial, special sequence to calculate the lowest ordered value in the recursion. A "kernel" of operation **420** that can be looped over multiple values of j for the first partial product calculation is presented immediately below.

The third instruction **413** VPMUL_HI calculates the first sub partial product term (A[0]*B[0]; j=0) and returns its higher half (Hi_0) in resultant register R_Hi. A fourth instruction VPMUL_LO **414** calculates the second sub partial product term (A[0]*B[1]; j=1) and returns its lower half (Lo_1) in resultant register R_Lo. A fifth instruction **415** performs aligned addition by adding the contents of R_Lo, R_Hi and their corresponding (aligned) element/digit in S (S[1]=0) and re-storing it in S.

Sequence **413**, **414** and **415** corresponds to a "kernel" **420** that can be looped for j=1 through 7. For example, continuing with the next j=2 incursion, a sixth instruction **416** VPMUL_HI calculates the second sub partial product term (A[0]*B[1]; j=1) and returns its higher half (Hi_1) in resultant register R_Hi. A seventh instruction VP_MUL **417** calculates the third sub partial product term (A[0]*B[2]; j=2) and returns its lower half (Lo_**2**) in resultant register R_Lo. An eighth instruction **418** performs aligned addition by adding the contents of R_Lo, R_Hi and their corresponding (aligned) element/digit in S (S[1]=0) and re-storing it in S.

The kernel can continue to loop through j=7. After the j=7 loop is performed, the digits in S have been calculated through element S[7]. A final sequence to complete the recursion for the first partial product is to execute a last VPMUL_HI instruction **421** which calculates the eighth sub partial product term (A[0]*B[7]; j=7) and returns its higher half (Hi_7) to R_Hi, and, execute a last instruction **422** that performs an aligned add 423 of the contents of R_Hi with the highest ordered digit in S (S[8]) and re-storing the result in S. At this point, S contains the first partial product.

Each subsequent partial product can then be calculated as substantially as described above. Two noteworthy features are the initial value of S will no longer be zero but rather contain an accumulation of the previously calculated partial products. Also, the alignment of each partial product needs to move one digit to the left as compared to the previously calculated partial product (similar to the alignment relationship of staircase structures in FIG. **3***b*).

Note that S is a nine element vector. That is, S has nine 64 bit values to represent the accumulated partial product terms. In an embodiment where the maximum vector size is 512 bits and digits of S are represented by 64 bit values, the size of S exceeds 512 bits by 128 bits. As such, two vectors S1 and S2 may be used by the instruction sequence where S1 keeps elements S[7:0] and S2 keeps S[8]. In this case, S1 is read from//written to for all instructions described above except instructions **425** and **427** which write to S2.

FIG. **5***a* shows another approach having a different operational pattern in the kernel. As will be described in more detail below, the repeatable kernel of the approach of FIG. **5***a* includes two ADD instructions to help accumulate terms for neighboring elements in S.

For the initial j=0 recursion, a VPMUL_Lo instruction is executed **511** to determine the lower half of A[0]*B[0] (Lo_0) and the resultant is stored in R_Lo, and, a VPMUL_Hi instruction is executed **512** to determine the upper half of A[0]*B[0] (Hi_0) and the resultant is stored in R_Hi. An ADD instruction **513** then adds the S[0] term (which is initially zero as is all digits of S for the initial j=0 recursion) to the R_Lo value and stored back in S[0]. Another ADD instruction **514** adds the S[1] term to the R_Hi value and the result is stored back in S[1].

For the next, j=1 recursion, again VPMUL_Lo and VPMUL_Hi instructions are executed **515**, **516** with respec-

tive results being stored in R_Lo and R_Hi respectively. A first subsequent ADD instruction adds 517 the contents of S[j]=S[1] to the contents of R_Lo and stores the result back in S[j]=S[1]. A second subsequent ADD instruction adds 518 the contents of S[j+1]=S[2] to the contents of R_Hi and stores the result back in S[j+1]=S[2].

Steps **511** through **514** (or **515** through **518**) correspond to a kernel that is repeated for each of the following recursions for j=2 through j=7. At the end of the j=7 cycle, each of digits S[2] through S[8] have been written to, which, corresponds to the partial product of A[0]*B[7:0]. The same sequence as described above for the A[0] multiplier is then repeated for each of A[1] through A[7]. Here, the accumulated partial product(s) of the previously determined partial product are updated/accumulated in S. The alignment of each subsequent iteration for a multiplier term should be aligned one digit to the left as compared to the alignment of the recursion performed for the preceding multiplier term.

Other recursion patterns than those presented in FIGS. **4***a* and **5***a* may be possible. FIGS. **4***a* and **5***a* also may utilize a unique approach with respect to the handling of the carry terms of the various ADD operations. Specifically, mask vector register space may be used to handle any mathematical carries that may be ancillary to the resultant of an ADD instruction.

FIG. **4***b* shows a more detailed implementation of an embodiment of the kernel **420** of FIG. **4***a*. With respect to the approach of FIG. **4***b*, the ADD instructions observed therein include an additional input k which corresponds to a mask register that is used to keep carry terms. Here, any carry term to be incorporated into the addition of the ADD instruction is received through mask register k and any carry term generated from the addition is "written back" to the mask register k. That is, mask register k is specified as containing both a source operand **430** and resultant **431**. As envisioned, the source operand k **430** holds the carry term from the ADD instruction of the immediately preceding recursion. The carry term is added into the addition performed by ADD instruction **432**. Any carry term that is generated from the addition performed by ADD instruction **432** is stored back into k as the resultant carry term **431** for use by the ADD instruction of the immediately following recursion.

A mathematical artifact of adding three operands is that the carry term may be larger than one bit. For example, if three 64 bit operands are added, the result may be 66 bits wide. As such, in this case, the carry term may be two bits rather than one bit. In an embodiment, rather than numerically add these carry terms in the ADD instruction of the next recursion, the carry terms are simply "written" as the least significant bits of the summation resultant. That is, the logic circuitry that implements ADD instruction **432** is designed to write the contents of the k source operand **430** as the lowest ordered bits of the ADD resultant (not the carry resultant **431**) that is stored in S.

The approach of FIG. **5***a* does not utilize a "three input operand" ADD instruction. Instead, a two input operand ADD instruction is used. Nevertheless, three terms are being added in each recursion. As such, the mathematical artifact referred to just above still applies. That is, at least for 64 bit digits, the addition performed to completely calculate each S[j] term may mathematically generate a two bit carry term. In order to address this feature, two different carry terms k0, k1 are separately tracked in mask register space as observed in the more detailed recursion flow of FIG. **5***b*.

Essentially, as any addition may generate a carry term for the "next addition to the left", as long as carry terms are forwarded in this manner the mathematical results will be accurate. Careful observation of the instruction flow reveals that both of the resultant k0, k1 carry terms are used as source operands for their respective "next addition to the left".

Note that in the case where the instruction sequences of FIGS. **4***a*, **4***b*, **5***a*, **5***b* are performed on a vector processor having 512 bit input operands which can be granularized to eight elements of 64 bits per element, the instruction sequences of FIGS. **4***a*, **4***b*, **5***a* and **5***b* are capable of supporting a procedure that simultaneously multiplies eight large multiplicands by eight respective large multipliers. That is, for example, a first input vector may be created having 8 64 bit elements where each element corresponds to a specific digit in eight different multiplicands, and, a second input input vector may be created having 8 64 bit elements where each element corresponds to a specific digit in eight different multipliers. With these and similarly structured vectors, the operations observed in FIGS. **4***a*, **4***b*, **5***a* and **5***b* can simultaneously multiply eight multiplicand and multiplier pairs.

FIG. **4***c* shows a logic design for an execution unit that can perform the VPMUL_LO and VPMUL_HI instructions as described above. The logic design of FIG. **4***c* can be used to support the multiplication instructions of FIG. **4***a*, **4***b*, **5***a* or **5***b*. As observed in FIG. **4***c*, a multiplier **450** receives a first input operand from a first input operand register **451** and receives a second input operand from a second input operand register **452**. Input operand registers **451**, **452** may be part of vector register space, an output of a data fetch stage of an instruction execution pipeline, or, an input of the execution unit. Multiplexer logic circuitry **453** selects either the low half or the right half of the full multiplication output. Whether the low half or right half is selected is determined from the instruction fetch and decode stage of the instruction execution pipeline (specifically, the decoding of the instruction opcode that specifies whether the instruction is VPMUL_LO or VPMUL_HI).

The selected half is presented to write mask circuitry **454**. A mask vector stored in mask vector register **455** is applied as an input to write mask circuitry **454**. Mask write circuitry **454** applies the mask to the selected half and the result is written to resultant register **456**. Resultant register **456** may be in vector register space or at the output of the execution unit. Additional features may be included to the base design of FIG. **4***c* such as support for different "digit" bit widths. In one embodiment, the granularity of the multiplier, the selection logic and the write mask circuitry is such that the digit width can be any size of 2^n provided it is equal to or less than maximum vector input operand size (e.g., 512 bits). For example, if n=4, the digit width is 16 bits which corresponds to a capability of simultaneously multiplying 32 different multiplicands and respective multipliers for a 512 bit input operand size.

FIG. **4***d* shows a logic design for a three input operand ADD instruction that uses mask register space to handle carry terms. The logic design of FIG. **4***d* can be used by an execution unit that supports the ADD instructions of FIGS. **4***a* and **4***b*. As observed in FIG. **4***d*, three input operands are respectively provided to an adder circuit **464** by way of input operand registers **461**, **462** and **463**. Input operand registers **461**, **462**, **463** may be from vector register space, an output of a data fetch stage of an instruction execution pipeline or an input of the execution unit. A mask input register **465** receives, potentially, mask vectors for other instructions supported by the execution unit. Consequently, outputs of the mask input register **465** flow to write mask circuitry **466**. The mask input register **465** may be part of vector register

space, an output of a data fetch stage or an input of the execution unit. To support the three input ADD instruction, however, the mask register **465** also supplies carry terms that are provided to the carry input of the adder **464**. Alternatively, as described above, signal lines carrying the carry input from register **465** may be directly routed to the lowest ordered bits of the resultant. A carry output from the adder **464** is provided to an output mask register **467** whose contents may write over whatever register sourced the carry terms in register **465**.

FIG. **5***c* shows a logic design for a two input operand ADD instruction that uses mask register space to handle carry terms. The logic design of FIG. **5***c* can be used by an execution unit that supports the ADD instructions of FIGS. **5***a* and **5***b*. As observed in FIG. **5***c*, two input operands are respectively provided to an adder circuit **564** by way of input operand registers **562** and **563**. Input operand registers **562**, **563** may be from vector register space, an output of a data fetch stage of an instruction execution pipeline or an input of the execution unit. A mask input register **565** receives, potentially, mask vectors for other instructions supported by the execution unit. Consequently, outputs of the mask input register **565** flow to write mask circuitry **566**. The mask input register **565** may be part of vector register space, an output of a data fetch stage or an input of the execution unit. To support the two input ADD instruction, however, the mask register **565** also supplies carry terms that are provided to the carry input of the adder **564**. A carry output from the adder **564** is provided to an output mask register **567** whose contents may write over whatever register sourced the carry terms in register **565**.

Exemplary Instruction Formats

Embodiments of the instruction(s) described herein may be embodied in different formats. For example, the instruction(s) described herein may be embodied as a VEX, generic vector friendly, or other format. Details of VEX and a generic vector friendly format are discussed below. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

VEX Instruction Format

VEX encoding allows instructions to have more than two operands, and allows SIMD vector registers to be longer than 128 bits. The use of a VEX prefix provides for three-operand (or more) syntax. For example, previous two-operand instructions performed operations such as A=A+B, which overwrites a source operand. The use of a VEX prefix enables operands to perform nondestructive operations such as A=B+C.

FIG. **6**A illustrates an exemplary AVX instruction format including a VEX prefix **602**, real opcode field **630**, Mod RIM byte **640**, SIB byte **650**, displacement field **662**, and IMM8 **672**. FIG. **6**B illustrates which fields from FIG. **6**A make up a full opcode field **674** and a base operation field **642**. FIG. **6**C illustrates which fields from FIG. **6**A make up a register index field **644**.

VEX Prefix (Bytes 0-2) **602** is encoded in a three-byte form. The first byte is the Format Field **640** (VEX Byte 0, bits [7:0]), which contains an explicit C4 byte value (the unique value used for distinguishing the C4 instruction format). The second-third bytes (VEX Bytes 1-2) include a number of bit fields providing specific capability. Specifically, REX field **605** (VEX Byte 1, bits [7-5]) consists of a VEX.R bit field (VEX Byte 1, bit [7]-R), VEX.X bit field (VEX byte 1, bit [6]-X), and VEX.B bit field (VEX byte 1, bit[5]-B). Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding VEX.R, VEX.X, and VEX.B. Opcode map field **615** (VEX byte 1, bits [4:0]-mmmmm) includes content to encode an implied leading opcode byte. W Field **664** (VEX byte 2, bit [7]-W)—is represented by the notation VEX.W, and provides different functions depending on the instruction. The role of VEX.vvvv **620** (VEX Byte 2, bits [6:3]-vvvv) may include the following: 1) VEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) VEX.vvvv encodes the destination register operand, specified in is complement form for certain vector shifts; or 3) VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. If VEX.L **668** Size field (VEX byte 2, bit [2]-L)=0, it indicates 128 bit vector; if VEX.L=1, it indicates 256 bit vector. Prefix encoding field **625** (VEX byte 2, bits [1:0]-pp) provides additional bits for the base operation field.

Real Opcode Field **630** (Byte 3) is also known as the opcode byte. Part of the opcode is specified in this field.

MOD R/M Field **640** (Byte 4) includes MOD field **642** (bits [7-6]), Reg field **644** (bits [5-3]), and R/M field **646** (bits [2-0]). The role of Reg field **644** may include the following: encoding either the destination register operand or a source register operand (the rrr of Rrrr), or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field **646** may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB)—The content of Scale field **650** (Byte 5) includes SS**652** (bits [7-6]), which is used for memory address generation. The contents of SIB.xxx **654** (bits [5-3]) and SIB.bbb **656** (bits [2-0]) have been previously referred to with regard to the register indexes Xxxx and Bbbb.

The Displacement Field **662** and the immediate field (IMM8) **672** contain address data.

Generic Vector Friendly Instruction Format

A vector friendly instruction format is an instruction format that is suited for vector instructions (e.g., there are certain fields specific to vector operations). While embodiments are described in which both vector and scalar operations are supported through the vector friendly instruction format, alternative embodiments use only vector operations the vector friendly instruction format.

FIGS. **7**A-**7**B are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention. FIG. **7**A is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention; while FIG. **7**B is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention. Specifically, a generic vector friendly instruction format **700** for which are defined class A and class B instruction templates, both of which include no memory access **705** instruction templates and memory access **720** instruction templates. The term generic in the context of the vector friendly instruction format refers to the instruction format not being tied to any specific instruction set.

While embodiments of the invention will be described in which the vector friendly instruction format supports the following: a 64 byte vector operand length (or size) with 32 bit (4 byte) or 64 bit (8 byte) data element widths (or sizes)

(and thus, a 64 byte vector consists of either 16 doubleword-size elements or alternatively, 8 quadword-size elements); a 64 byte vector operand length (or size) with 16 bit (2 byte) or 8 bit (1 byte) data element widths (or sizes); a 32 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); and a 16 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); alternative embodiments may support more, less and/or different vector operand sizes (e.g., 256 byte vector operands) with more, less, or different data element widths (e.g., 128 bit (16 byte) data element widths).

The class A instruction templates in FIG. **7**A include: 1) within the no memory access **705** instruction templates there is shown a no memory access, full round control type operation **710** instruction template and a no memory access, data transform type operation **715** instruction template; and 2) within the memory access **720** instruction templates there is shown a memory access, temporal **725** instruction template and a memory access, non-temporal **730** instruction template. The class B instruction templates in FIG. **7**B include: 1) within the no memory access **705** instruction templates there is shown a no memory access, write mask control, partial round control type operation **712** instruction template and a no memory access, write mask control, vsize type operation **717** instruction template; and 2) within the memory access **720** instruction templates there is shown a memory access, write mask control **727** instruction template.

The generic vector friendly instruction format **700** includes the following fields listed below in the order illustrated in FIGS. **7**A-**7**B. In conjunction with the discussions above of FIGS. **4***a,b,c,d* and **5***,a,b,c* in an embodiment, referring to the format details provided below in FIGS. **7**A-B and **8**, either a non memory access instruction type **705** or a memory access instruction type **720** may be utilized. Addresses for the read mask(s), input vector operand(s) and destination may be identified in register address field **744** described below. In a further embodiment, the write mask is specified in write mask field **770**.

Format field **740**—a specific value (an instruction format identifier value) in this field uniquely identifies the vector friendly instruction format, and thus occurrences of instructions in the vector friendly instruction format in instruction streams. As such, this field is optional in the sense that it is not needed for an instruction set that has only the generic vector friendly instruction format.

Base operation field **742**—its content distinguishes different base operations.

Register index field **744**—its content, directly or through address generation, specifies the locations of the source and destination operands, be they in registers or in memory. These include a sufficient number of bits to select N registers from a P×Q (e.g. 32×512, 16×128, 32×1024, 64×1024) register file. While in one embodiment N may be up to three sources and one destination register, alternative embodiments may support more or less sources and destination registers (e.g., may support up to two sources where one of these sources also acts as the destination, may support up to three sources where one of these sources also acts as the destination, may support up to two sources and one destination).

Modifier field **746**—its content distinguishes occurrences of instructions in the generic vector instruction format that specify memory access from those that do not; that is, between no memory access **705** instruction templates and memory access **720** instruction templates. Memory access operations read and/or write to the memory hierarchy (in some cases specifying the source and/or destination addresses using values in registers), while non-memory access operations do not (e.g., the source and destinations are registers). While in one embodiment this field also selects between three different ways to perform memory address calculations, alternative embodiments may support more, less, or different ways to perform memory address calculations.

Augmentation operation field **750**—its content distinguishes which one of a variety of different operations to be performed in addition to the base operation. This field is context specific. In one embodiment of the invention, this field is divided into a class field **768**, an alpha field **752**, and a beta field **754**. The augmentation operation field **750** allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

Scale field **760**—its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses 2^{scale}*index+base).

Displacement Field **762**A—its content is used as part of memory address generation (e.g., for address generation that uses 2^{scale}*index+base+displacement).

Displacement Factor Field **762**B (note that the juxtaposition of displacement field **762**A directly over displacement factor field **762**B indicates one or the other is used)—its content is used as part of address generation; it specifies a displacement factor that is to be scaled by the size of a memory access (N)—where N is the number of bytes in the memory access (e.g., for address generation that uses 2^{scale}*index+base+scaled displacement). Redundant low-order bits are ignored and hence, the displacement factor field's content is multiplied by the memory operands total size (N) in order to generate the final displacement to be used in calculating an effective address. The value of N is determined by the processor hardware at runtime based on the full opcode field **774** (described later herein) and the data manipulation field **754**C. The displacement field **762**A and the displacement factor field **762**B are optional in the sense that they are not used for the no memory access **705** instruction templates and/or different embodiments may implement only one or none of the two.

Data element width field **764**—its content distinguishes which one of a number of data element widths is to be used (in some embodiments for all instructions; in other embodiments for only some of the instructions). This field is optional in the sense that it is not needed if only one data element width is supported and/or data element widths are supported using some aspect of the opcodes.

Write mask field **770**—its content controls, on a per data element position basis, whether that data element position in the destination vector operand reflects the result of the base operation and augmentation operation. Class A instruction templates support merging-writemasking, while class B instruction templates support both merging- and zeroing-writemasking. When merging, vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation (specified by the base operation and the augmentation operation); in other one embodiment, preserving the old value of each element of the destination where the corresponding mask bit has a 0. In contrast, when zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation (specified by the base operation and the augmentation operation); in one embodiment, an element of the destination is set to 0 when the corresponding mask bit has a 0 value. A subset of this functionality is the ability to

control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive. Thus, the write mask field **770** allows for partial vector operations, including loads, stores, arithmetic, logical, etc. While embodiments of the invention are described in which the write mask field's **770** content selects one of a number of write mask registers that contains the write mask to be used (and thus the write mask field's **770** content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's **770** content to directly specify the masking to be performed.

Immediate field **772**—its content allows for the specification of an immediate. This field is optional in the sense that is it not present in an implementation of the generic vector friendly format that does not support immediate and it is not present in instructions that do not use an immediate.

Class field **768**—its content distinguishes between different classes of instructions. With reference to FIGS. **7**A-B, the contents of this field select between class A and class B instructions. In FIGS. **7**A-B, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A **768**A and class B **768**B for the class field **768** respectively in FIGS. **7**A-B).

Instruction Templates of Class A

In the case of the non-memory access **705** instruction templates of class A, the alpha field **752** is interpreted as an RS field **752**A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round **752**A.**1** and data transform **752**A.**2** are respectively specified for the no memory access, round type operation **710** and the no memory access, data transform type operation **715** instruction templates), while the beta field **754** distinguishes which of the operations of the specified type is to be performed. In the no memory access **705** instruction templates, the scale field **760**, the displacement field **762**A, and the displacement scale filed **762**B are not present.

No-Memory Access Instruction Templates—Full Round Control Type Operation

In the no memory access full round control type operation **710** instruction template, the beta field **754** is interpreted as a round control field **754**A, whose content(s) provide static rounding. While in the described embodiments of the invention the round control field **754**A includes a suppress all floating point exceptions (SAE) field **756** and a round operation control field **758**, alternative embodiments may support may encode both these concepts into the same field or only have one or the other of these concepts/fields (e.g., may have only the round operation control field **758**).

SAE field **756**—its content distinguishes whether or not to disable the exception event reporting; when the SAE field's **756** content indicates suppression is enabled, a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler.

Round operation control field **758**—its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field **758** allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's **750** content overrides that register value.

No Memory Access Instruction Templates—Data Transform Type Operation

In the no memory access data transform type operation **715** instruction template, the beta field **754** is interpreted as a data transform field **754**B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

In the case of a memory access **720** instruction template of class A, the alpha field **752** is interpreted as an eviction hint field **752**B, whose content distinguishes which one of the eviction hints is to be used (in FIG. **7**A, temporal **752**B.**1** and non-temporal **752**B.**2** are respectively specified for the memory access, temporal **725** instruction template and the memory access, non-temporal **730** instruction template), while the beta field **754** is interpreted as a data manipulation field **754**C, whose content distinguishes which one of a number of data manipulation operations (also known as primitives) is to be performed (e.g., no manipulation; broadcast; up conversion of a source; and down conversion of a destination). The memory access **720** instruction templates include the scale field **760**, and optionally the displacement field **762**A or the displacement scale field **762**B.

Vector memory instructions perform vector loads from and vector stores to memory, with conversion support. As with regular vector instructions, vector memory instructions transfer data from/to memory in a data element-wise fashion, with the elements that are actually transferred is dictated by the contents of the vector mask that is selected as the write mask.

Memory Access Instruction Templates—Temporal

Temporal data is data likely to be reused soon enough to benefit from caching. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Memory Access Instruction Templates—Non-Temporal

Non-temporal data is data unlikely to be reused soon enough to benefit from caching in the 1st-level cache and should be given priority for eviction. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Instruction Templates of Class B

In the case of the instruction templates of class B, the alpha field **752** is interpreted as a write mask control (Z) field **752**C, whose content distinguishes whether the write masking controlled by the write mask field **770** should be a merging or a zeroing.

In the case of the non-memory access **705** instruction templates of class B, part of the beta field **754** is interpreted as an RL field **757**A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round **757**A.**1** and vector length (VSIZE) **757**A.**2** are respectively specified for the no memory access, write mask control, partial round control type operation **712** instruction template and the no memory access, write mask control, VSIZE type operation **717** instruction template), while the rest of the beta field **754** distinguishes which of the operations of the specified type is to be performed. In the no memory access **705** instruction templates, the scale field **760**, the displacement field **762**A, and the displacement scale filed **762**B are not present.

In the no memory access, write mask control, partial round control type operation **710** instruction template, the rest of the beta field **754** is interpreted as a round operation field **759**A and exception event reporting is disabled (a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler).

Round operation control field **759**A—just as round operation control field **758**, its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field **759**A allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's **750** content overrides that register value.

In the no memory access, write mask control, VSIZE type operation **717** instruction template, the rest of the beta field **754** is interpreted as a vector length field **759**B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

In the case of a memory access **720** instruction template of class B, part of the beta field **754** is interpreted as a broadcast field **757**B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field **754** is interpreted the vector length field **759**B. The memory access **720** instruction templates include the scale field **760**, and optionally the displacement field **762**A or the displacement scale field **762**B.

With regard to the generic vector friendly instruction format **700**, a full opcode field **774** is shown including the format field **740**, the base operation field **742**, and the data element width field **764**. While one embodiment is shown where the full opcode field **774** includes all of these fields, the full opcode field **774** includes less than all of these fields in embodiments that do not support all of them. The full opcode field **774** provides the operation code (opcode).

The augmentation operation field **750**, the data element width field **764**, and the write mask field **770** allow these features to be specified on a per instruction basis in the generic vector friendly instruction format.

The combination of write mask field and data element width field create typed instructions in that they allow the mask to be applied based on different data element widths.

The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments of the invention, different processors or different cores within a processor may support only class A, only class B, or both classes. For instance, a high performance general purpose out-of-order core intended for general-purpose computing may support only class B, a core intended primarily for graphics and/or scientific (throughput) computing may support only class A, and a core intended for both may support both (of course, a core that has some mix of templates and instructions from both classes but not all templates and instructions from both classes is within the purview of the invention). Also, a single processor may include multiple cores, all of which support the same class or in which different cores support different class. For instance, in a processor with separate graphics and general purpose cores, one of the graphics cores intended primarily for graphics and/or scientific computing may support only class A, while one or more of the general purpose cores may be high performance general purpose cores with out of order execution and register renaming intended for general-purpose computing that support only class B. Another processor that does not have a separate graphics core, may include one more general purpose in-order or out-of-order cores that support both class A and class B. Of course, features from one class may also be implement in the other class in different embodiments of the invention. Programs written in a high level language would be put (e.g., just in time compiled or statically compiled) into an variety of different executable forms, including: 1) a form having only instructions of the class(es) supported by the target processor for execution; or 2) a form having alternative routines written using different combinations of the instructions of all classes and having control flow code that selects the routines to execute based on the instructions supported by the processor which is currently executing the code.

Exemplary Specific Vector Friendly Instruction Format

FIG. **8** is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. FIG. **8** shows a specific vector friendly instruction format **800** that is specific in the sense that it specifies the location, size, interpretation, and order of the fields, as well as values for some of those fields. The specific vector friendly instruction format **800** may be used to extend the x86 instruction set, and thus some of the fields are similar or the same as those used in the existing x86 instruction set and extension thereof (e.g., AVX). This format remains consistent with the prefix encoding field, real opcode byte field, MOD R/M field, SIB field, displacement field, and immediate fields of the existing x86 instruction set with extensions. The fields from FIG. **7** into which the fields from FIG. **8** map are illustrated.

It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format **800** in the context of the generic vector friendly instruction format **700** for illustrative purposes, the invention is not limited to the specific vector friendly instruction format **800** except where claimed. For example, the generic vector friendly instruction format **700** contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format **800** is shown as having fields of specific sizes. By way of specific example, while the data element width field **764** is illustrated as a one bit field in the specific vector friendly instruction format **800**, the invention is not so limited (that is, the generic vector friendly instruction format **700** contemplates other sizes of the data element width field **764**).

The generic vector friendly instruction format **700** includes the following fields listed below in the order illustrated in FIG. **8**A.

EVEX Prefix (Bytes 0-3) **802**—is encoded in a four-byte form.

Format Field **740** (EVEX Byte 0, bits [7:0])—the first byte (EVEX Byte 0) is the format field **740** and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention).

The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

REX field **805** (EVEX Byte 1, bits [7-5])—consists of a EVEX.R bit field (EVEX Byte 1, bit [7]-R), EVEX.X bit field (EVEX byte 1, bit [6]-X), and 757BEX byte 1, bit[5]-B). The EVEX.R, EVEX.X, and EVEX.B bit fields provide the same functionality as the corresponding VEX bit fields, and are encoded using is complement form, i.e. ZMM0 is encoded as 1111B, ZMM15 is encoded as 0000B. Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding EVEX.R, EVEX.X, and EVEX.B.

REX' field **710**—this is the first part of the REX' field **710** and is the EVEX.R' bit field (EVEX Byte 1, bit [4]-R') that is used to encode either the upper 16 or lower 16 of the extended 32 register set. In one embodiment of the inven-

tion, this bit, along with others as indicated below, is stored in bit inverted format to distinguish (in the well-known x86 32-bit mode) from the BOUND instruction, whose real opcode byte is 62, but does not accept in the MOD R/M field (described below) the value of 11 in the MOD field; alternative embodiments of the invention do not store this and the other indicated bits below in the inverted format. A value of 1 is used to encode the lower 16 registers. In other words, R'Rrrr is formed by combining EVEX.R', EVEX.R, and the other RRR from other fields.

Opcode map field **815** (EVEX byte 1, bits [3:0]-mmmm)—its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

Data element width field **764** (EVEX byte 2, bit [7]-W)—is represented by the notation EVEX.W. EVEX.W is used to define the granularity (size) of the datatype (either 32-bit data elements or 64-bit data elements).

EVEX.vvvv **820** (EVEX Byte 2, bits [6:3]-vvvv)—the role of EVEX.vvvv may include the following: 1) EVEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) EVEX.vvvv encodes the destination register operand, specified in is complement form for certain vector shifts; or 3) EVEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. Thus, EVEX.vvvv field **820** encodes the 4 low-order bits of the first source register specifier stored in inverted (1s complement) form. Depending on the instruction, an extra different EVEX bit field is used to extend the specifier size to 32 registers.

EVEX.U **768** Class field (EVEX byte 2, bit [2]-U)—If EVEX.U=0, it indicates class A or EVEX.U0; if EVEX.U=1, it indicates class B or EVEX.U1.

Prefix encoding field **825** (EVEX byte 2, bits [1:0]-pp)—provides additional bits for the base operation field. In addition to providing support for the legacy SSE instructions in the EVEX prefix format, this also has the benefit of compacting the SIMD prefix (rather than requiring a byte to express the SIMD prefix, the EVEX prefix requires only 2 bits). In one embodiment, to support legacy SSE instructions that use a SIMD prefix (66H, F2H, F3H) in both the legacy format and in the EVEX prefix format, these legacy SIMD prefixes are encoded into the SIMD prefix encoding field; and at runtime are expanded into the legacy SIMD prefix prior to being provided to the decoder's PLA (so the PLA can execute both the legacy and EVEX format of these legacy instructions without modification). Although newer instructions could use the EVEX prefix encoding field's content directly as an opcode extension, certain embodiments expand in a similar fashion for consistency but allow for different meanings to be specified by these legacy SIMD prefixes. An alternative embodiment may redesign the PLA to support the 2 bit SIMD prefix encodings, and thus not require the expansion.

Alpha field **752** (EVEX byte 3, bit [7]-EH; also known as EVEX.EH, EVEX.rs, EVEX.RL, EVEX.write mask control, and EVEX.N; also illustrated with α)—as previously described, this field is context specific.

Beta field **754** (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s_{2-0}, EVEX.r_{2-0}, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with βββ)—as previously described, this field is context specific.

REX' field **710**—this is the remainder of the REX' field and is the EVEX.V' bit field (EVEX Byte 3, bit [3]-V') that may be used to encode either the upper 16 or lower 16 of the extended 32 register set. This bit is stored in bit inverted format. A value of 1 is used to encode the lower 16 registers. In other words, V'VVVV is formed by combining EVEX.V', EVEX.vvvv.

Write mask field **770** (EVEX byte 3, bits [2:0]-kkk)—its content specifies the index of a register in the write mask registers as previously described. In one embodiment of the invention, the specific value EVEX.kkk=000 has a special behavior implying no write mask is used for the particular instruction (this may be implemented in a variety of ways including the use of a write mask hardwired to all ones or hardware that bypasses the masking hardware).

Real Opcode Field **830** (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field.

MOD R/M Field **840** (Byte 5) includes MOD field **842**, Reg field **844**, and R/M field **846**. As previously described, the MOD field's **842** content distinguishes between memory access and non-memory access operations. The role of Reg field **844** can be summarized to two situations: encoding either the destination register operand or a source register operand, or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field **846** may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB) Byte (Byte 6)—As previously described, the scale field's **750** content is used for memory address generation. SIB.xxx **854** and SIB.bbb **856**—the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

Displacement field **762**A (Bytes 7-10)—when MOD field **842** contains 10, bytes 7-10 are the displacement field **762**A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

Displacement factor field **762**B (Byte 7)—when MOD field **842** contains 01, byte 7 is the displacement factor field **762**B. The location of this field is that same as that of the legacy x86 instruction set 8-bit displacement (disp8), which works at byte granularity. Since disp8 is sign extended, it can only address between −128 and 127 bytes offsets; in terms of 64 byte cache lines, disp8 uses 8 bits that can be set to only four really useful values −128, −64, 0, and 64; since a greater range is often needed, disp32 is used; however, disp32 requires 4 bytes. In contrast to disp8 and disp32, the displacement factor field **762**B is a reinterpretation of disp8; when using displacement factor field **762**B, the actual displacement is determined by the content of the displacement factor field multiplied by the size of the memory operand access (N). This type of displacement is referred to as disp8*N. This reduces the average instruction length (a single byte of used for the displacement but with a much greater range). Such compressed displacement is based on the assumption that the effective displacement is multiple of the granularity of the memory access, and hence, the redundant low-order bits of the address offset do not need to be encoded. In other words, the displacement factor field **762**B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field **762**B is encoded the same way as an x86 instruction set 8-bit displacement (so no changes in the ModRM/SIB encoding rules) with the only exception that disp8 is overloaded to disp8*N. In other words, there are no changes in the encoding rules or encoding lengths but only in the interpretation of the displacement

value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset).

Immediate field **772** operates as previously described.

Full Opcode Field

FIG. **8**B is a block diagram illustrating the fields of the specific vector friendly instruction format **800** that make up the full opcode field **774** according to one embodiment of the invention. Specifically, the full opcode field **774** includes the format field **740**, the base operation field **742**, and the data element width (W) field **764**. The base operation field **742** includes the prefix encoding field **825**, the opcode map field **815**, and the real opcode field **830**.

Register Index Field

FIG. **8**C is a block diagram illustrating the fields of the specific vector friendly instruction format **800** that make up the register index field **744** according to one embodiment of the invention. Specifically, the register index field **744** includes the REX field **805**, the REX' field **810**, the MODR/M.reg field **844**, the MODR/M.r/m field **846**, the VVVV field **820**, xxx field **854**, and the bbb field **856**.

Augmentation Operation Field

FIG. **8**D is a block diagram illustrating the fields of the specific vector friendly instruction format **800** that make up the augmentation operation field **750** according to one embodiment of the invention. When the class (U) field **768** contains 0, it signifies EVEX.U0 (class A **768**A); when it contains 1, it signifies EVEX.U1 (class B **768**B). When U=0 and the MOD field **842** contains 11 (signifying a no memory access operation), the alpha field **752** (EVEX byte 3, bit [7]-EH) is interpreted as the rs field **752**A. When the rs field **752**A contains a 1 (round **752**A.**1**), the beta field **754** (EVEX byte 3, bits [6:4]-SSS) is interpreted as the round control field **754**A. The round control field **754**A includes a one bit SAE field **756** and a two bit round operation field **758**. When the rs field **752**A contains a 0 (data transform **752**A.**2**), the beta field **754** (EVEX byte 3, bits [6:4]-SSS) is interpreted as a three bit data transform field **754**B. When U=0 and the MOD field **842** contains 00, 01, or 10 (signifying a memory access operation), the alpha field **752** (EVEX byte 3, bit [7]-EH) is interpreted as the eviction hint (EH) field **752**B and the beta field **754** (EVEX byte 3, bits [6:4]-SSS) is interpreted as a three bit data manipulation field **754**C.

When U=1, the alpha field **752** (EVEX byte 3, bit [7]-EH) is interpreted as the write mask control (Z) field **752**C. When U=1 and the MOD field **842** contains 11 (signifying a no memory access operation), part of the beta field **754** (EVEX byte 3, bit [4]-S_0) is interpreted as the RL field **757**A; when it contains a 1 (round **757**A.**1**) the rest of the beta field **754** (EVEX byte 3, bit [6-5]-S_{2-1}) is interpreted as the round operation field **759**A, while when the RL field **757**A contains a 0 (VSIZE **757**.A**2**) the rest of the beta field **754** (EVEX byte 3, bit [6-5]-S_{2-1}) is interpreted as the vector length field **759**B (EVEX byte 3, bit [6-5]-L_{1-0}). When U=1 and the MOD field **842** contains 00, 01, or 10 (signifying a memory access operation), the beta field **754** (EVEX byte 3, bits [6:4]-SSS) is interpreted as the vector length field **759**B (EVEX byte 3, bit [6-5]-L_{1-0}) and the broadcast field **757**B (EVEX byte 3, bit [4]-B).

Exemplary Register Architecture

FIG. **9** is a block diagram of a register architecture **900** according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers **910** that are 512 bits wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymm0-16. The lower order 128 bits of the lower 16 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15. The specific vector friendly instruction format **800** operates on these overlaid register file as illustrated in the below tables.

Adjustable Vector Length	Class	Operations	Registers
Instruction Templates that do not include the vector length field 759B	A (FIG. 7A; U = 0)	710, 715, 725, 730	zmm registers (the vector length is 64 byte)
	B (FIG. 7B; U = 1)	712	zmm registers (the vector length is 64 byte)
Instruction Templates that do include the vector length field 759B	B (FIG. 7B; U = 1)	717, 727	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 759B

In other words, the vector length field **759**B selects between a maximum length and one or more other shorter lengths, where each such shorter length is half the length of the preceding length; and instructions templates without the vector length field **759**B operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format **800** operate on packed or scalar single/double-precision floating point data and packed or scalar integer data. Scalar operations are operations performed on the lowest order data element position in an zmm/ymm/xmm register; the higher order data element positions are either left the same as they were prior to the instruction or zeroed depending on the embodiment.

Write mask registers **915**—in the embodiment illustrated, there are 8 write mask registers (k0 through k7), each 64 bits in size. In an alternate embodiment, the write mask registers **915** are 16 bits in size. As previously described, in one embodiment of the invention, the vector mask register k0 cannot be used as a write mask; when the encoding that would normally indicate k0 is used for a write mask, it selects a hardwired write mask of 0xFFFF, effectively disabling write masking for that instruction.

General-purpose registers **925**—in the embodiment illustrated, there are sixteen 64-bit general-purpose registers that are used along with the existing x86 addressing modes to address memory operands. These registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

Scalar floating point stack register file (x87 stack) **945**, on which is aliased the MMX packed integer flat register file **950**—in the embodiment illustrated, the x87 stack is an eight-element stack used to perform scalar floating-point operations on 32/64/80-bit floating point data using the x87 instruction set extension; while the MMX registers are used to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

Exemplary Core Architectures, Processors, and Computer Architectures

Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a CPU including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include: 1) the coprocessor on a separate chip from the CPU; 2) the coprocessor on a separate die in the same package as a CPU; 3) the coprocessor on the same die as a CPU (in which case, such a coprocessor is sometimes referred to as special purpose logic, such as integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die the described CPU (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality. Exemplary core architectures are described next, followed by descriptions of exemplary processors and computer architectures.

Exemplary Core Architectures

In-Order and Out-of-Order Core Block Diagram

FIG. **10**A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. FIG. **10**B is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes in FIGS. **10**A-B illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

In FIG. **10**A, a processor pipeline **1000** includes a fetch stage **1002**, a length decode stage **1004**, a decode stage **1006**, an allocation stage **1008**, a renaming stage **1010**, a scheduling (also known as a dispatch or issue) stage **1012**, a register read/memory read stage **1014**, an execute stage **1016**, a write back/memory write stage **1018**, an exception handling stage **1022**, and a commit stage **1024**.

FIG. **10**B shows processor core **1090** including a front end unit **1030** coupled to an execution engine unit **1050**, and both are coupled to a memory unit **1070**. The core **1090** may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core, or a hybrid or alternative core type. As yet another option, the core **1090** may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

The front end unit **1030** includes a branch prediction unit **1032** coupled to an instruction cache unit **1034**, which is coupled to an instruction translation lookaside buffer (TLB) **1036**, which is coupled to an instruction fetch unit **1038**, which is coupled to a decode unit **1040**. The decode unit **1040** (or decoder) may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit **1040** may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core **1090** includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit **1040** or otherwise within the front end unit **1030**). The decode unit **1040** is coupled to a rename/allocator unit **1052** in the execution engine unit **1050**.

The execution engine unit **1050** includes the rename/allocator unit **1052** coupled to a retirement unit **1054** and a set of one or more scheduler unit(s) **1056**. The scheduler unit(s) **1056** represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) **1056** is coupled to the physical register file(s) unit(s) **1058**. Each of the physical register file(s) units **1058** represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. In one embodiment, the physical register file(s) unit **1058** comprises a vector registers unit, a write mask registers unit, and a scalar registers unit. These register units may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) unit(s) **1058** is overlapped by the retirement unit **1054** to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit **1054** and the physical register file(s) unit(s) **1058** are coupled to the execution cluster(s) **1060**. The execution cluster(s) **1060** includes a set of one or more execution units **1062** and a set of one or more memory access units **1064**. The execution units **1062** may perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions. The scheduler unit(s) **1056**, physical register file(s) unit(s) **1058**, and execution cluster(s) **1060** are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating point/packed integer/packed floating point/vector integer/vector floating point pipeline, and/or a memory access pipeline that each have their own scheduler unit, physical register file(s) unit, and/or execution cluster—and in the case of a separate memory access pipeline, certain embodiments are implemented in which only the execution cluster of this pipeline has the memory access unit(s) **1064**). It should also be understood

that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

The set of memory access units **1064** is coupled to the memory unit **1070**, which includes a data TLB unit **1072** coupled to a data cache unit **1074** coupled to a level 2 (L2) cache unit **1076**. In one exemplary embodiment, the memory access units **1064** may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit **1072** in the memory unit **1070**. The instruction cache unit **1034** is further coupled to a level 2 (L2) cache unit **1076** in the memory unit **1070**. The L2 cache unit **1076** is coupled to one or more other levels of cache and eventually to a main memory.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline **1000** as follows: 1) the instruction fetch **1038** performs the fetch and length decoding stages **1002** and **1004**; 2) the decode unit **1040** performs the decode stage **1006**; 3) the rename/allocator unit **1052** performs the allocation stage **1008** and renaming stage **1010**; 4) the scheduler unit(s) **1056** performs the schedule stage **1012**; 5) the physical register file(s) unit(s) **1058** and the memory unit **1070** perform the register read/memory read stage **1014**; the execution cluster **1060** perform the execute stage **1016**; 6) the memory unit **1070** and the physical register file(s) unit(s) **1058** perform the write back/memory write stage **1018**; 7) various units may be involved in the exception handling stage **1022**; and 8) the retirement unit **1054** and the physical register file(s) unit(s) **1058** perform the commit stage **1024**.

The core **1090** may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, Calif.; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, Calif.), including the instruction(s) described herein. In one embodiment, the core **1090** includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2, and/or some form of the generic vector friendly instruction format (U=0 and/or U=1) previously described), thereby allowing the operations used by many multimedia applications to be performed using packed data.

It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor also includes separate instruction and data cache units **1034**/**1074** and a shared L2 cache unit **1076**, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

Specific Exemplary in-Order Core Architecture

FIGS. **11**A-B illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip. The logic blocks communicate through a high-bandwidth interconnect network (e.g., a ring network) with some fixed function logic, memory I/O interfaces, and other necessary I/O logic, depending on the application.

FIG. **11**A is a block diagram of a single processor core, along with its connection to the on-die interconnect network **1102** and with its local subset of the Level 2 (L2) cache **1104**, according to embodiments of the invention. In one embodiment, an instruction decoder **1100** supports the x86 instruction set with a packed data instruction set extension. An L1 cache **1106** allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit **1108** and a vector unit **1110** use separate register sets (respectively, scalar registers **1112** and vector registers **1114**) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache **1106**, alternative embodiments of the invention may use a different approach (e.g., use a single register set or include a communication path that allow data to be transferred between the two register files without being written and read back).

The local subset of the L2 cache **1104** is part of a global L2 cache that is divided into separate local subsets, one per processor core. Each processor core has a direct access path to its own local subset of the L2 cache **1104**. Data read by a processor core is stored in its L2 cache subset **1104** and can be accessed quickly, in parallel with other processor cores accessing their own local L2 cache subsets. Data written by a processor core is stored in its own L2 cache subset **1104** and is flushed from other subsets, if necessary. The ring network ensures coherency for shared data. The ring network is bi-directional to allow agents such as processor cores, L2 caches and other logic blocks to communicate with each other within the chip. Each ring data-path is 1012-bits wide per direction.

FIG. **11**B is an expanded view of part of the processor core in FIG. **11**A according to embodiments of the invention. FIG. **11**B includes an L1 data cache **1106**A part of the L1 cache **1104**, as well as more detail regarding the vector unit **1110** and the vector registers **1114**. Specifically, the vector unit **1110** is a 16-wide vector processing unit (VPU) (see the 16-wide ALU **1128**), which executes one or more of integer, single-precision float, and double-precision float instructions. The VPU supports swizzling the register inputs with swizzle unit **1120**, numeric conversion with numeric convert units **1122**A-B, and replication with replication unit **1124** on the memory input. Write mask registers **1126** allow predicating resulting vector writes.

Processor with Integrated Memory Controller and Graphics

FIG. **12** is a block diagram of a processor **1200** that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention. The solid lined boxes in FIG. **12** illustrate a processor **1200** with a single core **1202**A, a system agent **1210**, a set of one or more bus controller units **1216**, while the optional addition of the dashed lined boxes illustrates an alternative processor **1200** with multiple cores **1202**A-N, a set of one or more integrated memory controller unit(s) **1214** in the system agent unit **1210**, and special purpose logic **1208**.

Thus, different implementations of the processor **1200** may include: 1) a CPU with the special purpose logic **1208** being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores **1202**A-N being one or more general purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, a combination of the two); 2) a coprocessor with the cores **1202**A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores **1202**A-N being a large number of general purpose in-order cores. Thus, the processor **1200** may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit), a high-throughput many integrated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor **1200** may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units **1206**, and external memory (not shown) coupled to the set of integrated memory controller units **1214**. The set of shared cache units **1206** may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof. While in one embodiment a ring based interconnect unit **1212** interconnects the integrated graphics logic **1208**, the set of shared cache units **1206**, and the system agent unit **1210**/integrated memory controller unit(s) **1214**, alternative embodiments may use any number of well-known techniques for interconnecting such units. In one embodiment, coherency is maintained between one or more cache units **1206** and cores **1202**-A-N.

In some embodiments, one or more of the cores **1202**A-N are capable of multi-threading. The system agent **1210** includes those components coordinating and operating cores **1202**A-N. The system agent unit **1210** may include for example a power control unit (PCU) and a display unit. The PCU may be or include logic and components needed for regulating the power state of the cores **1202**A-N and the integrated graphics logic **1208**. The display unit is for driving one or more externally connected displays.

The cores **1202**A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores **1202**A-N may be capable of execution the same instruction set, while others may be capable of executing only a subset of that instruction set or a different instruction set.

Exemplary Computer Architectures

FIGS. **13-16** are block diagrams of exemplary computer architectures. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

Referring now to FIG. **13**, shown is a block diagram of a system **1300** in accordance with one embodiment of the present invention. The system **1300** may include one or more processors **1310**, **1315**, which are coupled to a controller hub **1320**. In one embodiment the controller hub **1320** includes a graphics memory controller hub (GMCH) **1390** and an Input/Output Hub (IOH) **1350** (which may be on separate chips); the GMCH **1390** includes memory and graphics controllers to which are coupled memory **1340** and a coprocessor **1345**; the IOH **1350** is couples input/output (I/O) devices **1360** to the GMCH **1390**. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory **1340** and the coprocessor **1345** are coupled directly to the processor **1310**, and the controller hub **1320** in a single chip with the IOH **1350**.

The optional nature of additional processors **1315** is denoted in FIG. **13** with broken lines. Each processor **1310**, **1315** may include one or more of the processing cores described herein and may be some version of the processor **1200**.

The memory **1340** may be, for example, dynamic random access memory (DRAM), phase change memory (PCM), or a combination of the two. For at least one embodiment, the controller hub **1320** communicates with the processor(s) **1310**, **1315** via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection **1395**.

In one embodiment, the coprocessor **1345** is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like. In one embodiment, controller hub **1320** may include an integrated graphics accelerator.

There can be a variety of differences between the physical resources **1310**, **1315** in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

In one embodiment, the processor **1310** executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor **1310** recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor **1345**. Accordingly, the processor **1310** issues these coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor **1345**. Coprocessor(s) **1345** accept and execute the received coprocessor instructions.

Referring now to FIG. **14**, shown is a block diagram of a first more specific exemplary system **1400** in accordance with an embodiment of the present invention. As shown in FIG. **14**, multiprocessor system **1400** is a point-to-point interconnect system, and includes a first processor **1470** and a second processor **1480** coupled via a point-to-point interconnect **1450**. Each of processors **1470** and **1480** may be some version of the processor **1200**. In one embodiment of the invention, processors **1470** and **1480** are respectively processors **1310** and **1315**, while coprocessor **1438** is coprocessor **1345**. In another embodiment, processors **1470** and **1480** are respectively processor **1310** coprocessor **1345**.

Processors **1470** and **1480** are shown including integrated memory controller (IMC) units **1472** and **1482**, respectively. Processor **1470** also includes as part of its bus controller units point-to-point (P-P) interfaces **1476** and **1478**; similarly, second processor **1480** includes P-P interfaces **1486** and **1488**. Processors **1470**, **1480** may exchange information via a point-to-point (P-P) interface **1450** using P-P interface circuits **1478**, **1488**. As shown in FIG. **14**, IMCs **1472** and

1482 couple the processors to respective memories, namely a memory 1432 and a memory 1434, which may be portions of main memory locally attached to the respective processors.

Processors 1470, 1480 may each exchange information with a chipset 1490 via individual P-P interfaces 1452, 1454 using point to point interface circuits 1476, 1494, 1486, 1498. Chipset 1490 may optionally exchange information with the coprocessor 1438 via a high-performance interface 1439. In one embodiment, the coprocessor 1438 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

A shared cache (not shown) may be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

Chipset 1490 may be coupled to a first bus 1416 via an interface 1496. In one embodiment, first bus 1416 may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of the present invention is not so limited.

As shown in FIG. 14, various I/O devices 1414 may be coupled to first bus 1416, along with a bus bridge 1418 which couples first bus 1416 to a second bus 1420. In one embodiment, one or more additional processor(s) 1415, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor, are coupled to first bus 1416. In one embodiment, second bus 1420 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 1420 including, for example, a keyboard and/or mouse 1422, communication devices 1427 and a storage unit 1428 such as a disk drive or other mass storage device which may include instructions/code and data 1430, in one embodiment. Further, an audio I/O 1424 may be coupled to the second bus 1420. Note that other architectures are possible. For example, instead of the point-to-point architecture of FIG. 14, a system may implement a multi-drop bus or other such architecture.

Referring now to FIG. 15, shown is a block diagram of a second more specific exemplary system 1500 in accordance with an embodiment of the present invention. Like elements in FIGS. 14 and 15 bear like reference numerals, and certain aspects of FIG. 14 have been omitted from FIG. 15 in order to avoid obscuring other aspects of FIG. 15.

FIG. 15 illustrates that the processors 1470, 1480 may include integrated memory and I/O control logic ("CL") 1472 and 1482, respectively. Thus, the CL 1472, 1482 include integrated memory controller units and include I/O control logic. FIG. 15 illustrates that not only are the memories 1432, 1434 coupled to the CL 1472, 1482, but also that I/O devices 1514 are also coupled to the control logic 1472, 1482. Legacy I/O devices 1515 are coupled to the chipset 1490.

Referring now to FIG. 16, shown is a block diagram of a SoC 1600 in accordance with an embodiment of the present invention. Similar elements in FIG. 12 bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In FIG. 16, an interconnect unit(s) 1602 is coupled to: an application processor 1610 which includes a set of one or more cores 202A-N and shared cache unit(s) 1206; a system agent unit 1210; a bus controller unit(s) 1216; an integrated memory controller unit(s) 1214; a set or one or more coprocessors 1620 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 1630; a direct memory access (DMA) unit 1632; and a display unit 1640 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 1620 include a special-purpose processor, such as, for example, a network or communication processor, compression engine, GPGPU, a high-throughput MIC processor, embedded processor, or the like.

Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

Program code, such as code 1430 illustrated in FIG. 14, may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

The program code may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The program code may also be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as "IP cores" may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable's (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, cir-

cuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

Emulation (Including Binary Translation, Code Morphing, Etc.)

In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

FIG. **17** is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. FIG. **17** shows a program in a high level language **1702** may be compiled using an x86 compiler **1704** to generate x86 binary code **1706** that may be natively executed by a processor with at least one x86 instruction set core **1716**. The processor with at least one x86 instruction set core **1716** represents any processor that can perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one x86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. The x86 compiler **1704** represents a compiler that is operable to generate x86 binary code **1706** (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one x86 instruction set core **1716**. Similarly, FIG. **17** shows the program in the high level language **1702** may be compiled using an alternative instruction set compiler **1708** to generate alternative instruction set binary code **1710** that may be natively executed by a processor without at least one x86 instruction set core **1714** (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, Calif. and/or that execute the ARM instruction set of ARM Holdings of Sunnyvale, Calif.). The instruction converter **1712** is used to convert the x86 binary code **1706** into code that may be natively executed by the processor without an x86 instruction set core **1714**. This converted code is not likely to be the same as the alternative instruction set binary code **1710** because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter **1712** represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have an x86 instruction set processor or core to execute the x86 binary code **1706**.

What is claimed is:

1. A method comprising:

decoding a first instruction, a second instruction, a third instruction, a fourth instruction, and a fifth instruction with a hardware decoder of a hardware processor;

executing the first instruction with a hardware execution unit of the hardware processor to multiply a first input operand and a second input operand and present a lower portion of a result, said first input operand representing a first digit of a multiplier, said second input operand representing a first digit of a multiplicand;

executing the second instruction with the hardware execution unit of the hardware processor to multiply said first input operand and said second input operand and present an upper portion of a result;

executing the third instruction with the hardware execution unit of the hardware processor to multiply said first input operand and a third input operand and present a lower portion of a result, said third input operand representing a digit of said multiplicand that neighbors said first digit of said multiplicand;

executing the fourth instruction with the hardware execution unit of the hardware processor to multiply said first input operand and said third input operand and present an upper portion of a result; and

executing the fifth instruction with the hardware execution unit of the hardware processor to add aligned digits of the upper and lower portions and record a carry term in a mask register.

2. The method of claim **1** wherein the first and second instructions are executed in a same recursion.

3. The method of claim **1** wherein a multiplexer of said hardware execution unit of the hardware processor is to output a low half from a multiplier for the first instruction and a high half from the multiplier for the second instruction.

4. The method of claim **1** wherein the carry term is one of a plurality of carry terms that are separately tracked in mask register space.

5. The method of claim **1** wherein said executing the fifth instruction with the hardware execution unit of the hardware processor is to also add in an input carry term from the mask register.

6. The method of claim **1** wherein the carry term is more than one bit.

7. The method of claim **6** wherein said carry term is written as least significant bits of a next higher ordered accumulated partial product term.

8. A hardware processor comprising:

a hardware decoder to decode a first instruction, a second instruction, and an add instruction; and

a hardware execution unit to:

execute the first instruction to multiply a first input operand and a second input operand and present a lower portion of a result, said first and second input operands being respective elements of first and second input vectors,

execute the second instruction to multiply the first input operand and the second input operand and present an upper portion of a result, said first and second input operands being respective elements of first and second input vectors, and

execute the add instruction that is to add aligned digits of the upper and lower portions and cause a carry term of said add instruction's adding to be recorded in a mask register.

9. The hardware processor of claim 8 wherein said add instruction comprises an operand to identify the mask register.

10. The hardware processor of claim 8 wherein the carry term is a plurality of bits.

11. The hardware processor of claim 8 wherein said add instruction accepts an input carry term through said mask register.

12. The hardware processor of claim 11 wherein said add instruction writes said input carry term as least significant bits of its add resultant.

13. The hardware processor of claim 8 wherein a multiplexer of said hardware execution unit is to output a low half from a multiplier for the first instruction and a high half from the multiplier for the second instruction.

14. The hardware processor of claim 8 wherein said first and second instructions are vector instructions that multiply respective elements of first and second input vectors, said first input operand being an element of said first input vector and said second input operand being an element of a second input vector.

15. The hardware processor of claim 8, wherein the carry term is one of a plurality of carry terms that are separately tracked in mask register space.

16. A non-transitory machine readable medium containing program code that when processed by a processing unit causes a method to be performed, said method comprising:

decoding a first instruction, a second instruction, a third instruction, a fourth instruction, and a fifth instruction with a hardware decoder of a hardware processor;

executing the first instruction with a hardware execution unit of the hardware processor to multiply a first input operand and a second input operand and present a lower portion of a result, said first input operand representing a first digit of a multiplier, said second input operand representing a first digit of a multiplicand;

executing the second instruction with the hardware execution unit of the hardware processor to multiply said first input operand and said second input operand and present an upper portion of a result;

executing the third instruction with the hardware execution unit of the hardware processor to multiply said first input operand and a third input operand and present a lower portion of a result, said third input operand representing a digit of said multiplicand that neighbors said first digit of said multiplicand;

executing the fourth instruction with the hardware execution unit of the hardware processor to multiply said first input operand and said third input operand and present an upper portion of a result; and

executing the fifth instruction with the hardware execution unit of the hardware processor to add aligned digits of the upper and lower portions and record a carry term in a mask register.

17. The non-transitory machine readable medium of claim 16 wherein the first and second instructions are executed in a same recursion.

18. The non-transitory machine readable medium of claim 16 wherein a multiplexer of said hardware execution unit of the hardware processor is to output a low half from a multiplier for the first instruction and a high half from the multiplier for the second instruction.

19. The non-transitory machine readable medium of claim 16 wherein the carry term is one of a plurality of carry terms that are separately tracked in mask register space.

20. The non-transitory machine readable medium of claim 16 wherein said executing the fifth instruction with the hardware execution unit of the hardware processor is to also add in an input carry term from the mask register.

21. The non-transitory machine readable medium of claim 16 wherein the carry term is more than one bit.

22. The non-transitory machine readable medium of claim 21 wherein said carry term is an input carry term and said input carry term is written as least significant bits of said adding's resultant.

* * * * *